Reverse Engineering для начинающих



Денис Юричев

Reverse Engineering для начинающих

Денис Юричев
<dennis(a)yurichev.com>

Версия этого текста (10 октября 2015 г.).

Самая новая версия текста (а также англоязычная версия) доступна на сайте beginners.re. Версия для электронных читалок так же доступна на сайте.

Вы также можете подписаться на мой twitter для получения информации о новых версиях этого текста: @yurichev[1], либо подписаться на список рассылки[2].

Обложка нарисована Андреем Нечаевским: facebook.

[1] twitter.com/yurichev
[2] yurichev.com

Внимание: это сокращенная LITE-версия!

Она примерно в 6 раз короче полной версии (~150 страниц) и предназначена для тех, кто хочет краткого введения в основы reverse engineering. Здесь нет ничего о MIPS, ARM, OllyDBG, GCC, GDB, IDA, нет задач, примеров, и т.д.

Если вам всё ещё интересен reverse engineering, полная версия книги всегда доступна на моем сайте: beginners.re.

Оглавление

Предисловие

У термина «reverse engineering» несколько популярных значений: 1) исследование скомпилированных программ; 2) сканирование трехмерной модели для последующего копирования; 3) восстановление структуры СУБД. Настоящий сборник заметок связан с первым значением.

Об авторе



Денис Юричев — опытный reverse engineer и программист. С ним можно контактировать по емейлу: **dennis(a)yurichev.com**, или по Skype: **dennis.yurichev**.

Отзывы о книге *Reverse Engineering для начинающих*

- «It's very well done .. and for free .. amazing.»[3] Daniel Bilar, Siege Technologies, LLC.
- «... excellent and free»[4] Pete Finnigan, гуру по безопасности Oracle RDBMS.
- «... book is interesting, great job!» Michael Sikorski, автор книги *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- «... my compliments for the very nice tutorial!» Herbert Bos, профессор университета Vrije Universiteit Amsterdam, соавтор *Modern Operating Systems (4th Edition)*.
- «... It is amazing and unbelievable.» Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- «Thanks for the great work and your book.» Joris van de Vis, специалист по SAP Netweaver & Security.
- «... reasonable intro to some of the techniques.»[5] Mike Stay, преподаватель в Federal Law Enforcement Training Center, Georgia, US.
- «I love this book! I have several students reading it at the moment, plan to use it in graduate course.»[6] Сергей Братусь, Research Assistant Professor в отделе Computer Science в Dartmouth College
- «Dennis @Yurichev has published an impressive (and free!) book on reverse engineering»[7] Tanel Poder, эксперт по настройке производительности Oracle RDBMS.
- «This book is some kind of Wikipedia to beginners...» Archer, Chinese Translator, IT Security Researcher.
- «Прочел Вашу книгу — отличная работа, рекомендую на своих курсах студентам в качестве учебного пособия». Николай Ильин, преподаватель в ФТИ НТУУ «КПИ» и DefCon-UA

Благодарности

Тем, кто много помогал мне отвечая на массу вопросов: Андрей «herm1t» Баранович, Слава «Avid» Казаков.

Тем, кто присылал замечания об ошибках и неточностях: Станислав «Beaver» Бобрицкий, Александр Лысенко, Shell Rocket, Zhu Ruijin, Changmin Heo.

Просто помогали разными способами: Андрей Зубинский, Arnaud Patard (rtp на #debian-arm IRC), Александр Автаев.

Переводчикам на китайский язык: Antiy Labs (antiy.cn) и Archer.

[3]twitter.com/daniel_bilar/status/436578617221742593
[4]twitter.com/petefinnigan/status/400551705797869568
[5]reddit
[6]twitter.com/sergeybratus/status/505590326560833536
[7]twitter.com/TanelPoder/status/524668104065159169

Переводчику на корейский язык: Byungho Min.

Корректорам: Александр «Lstar» Черненький, Владимир Ботов, Андрей Бражук, Марк “Logxen” Купер, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen.

Васил Колев сделал очень много исправлений и указал на многие ошибки.

За иллюстрации и обложку: Андрей Нечаевский.

И ещё всем тем на github.com кто присылал замечания и исправления.

Было использовано множество пакетов LaTeX. Их авторов я также хотел бы поблагодарить.

Жертвователи

Те, кто поддерживал меня во время написании этой книги:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10).

Огромное спасибо каждому!

mini-ЧаВО

Q: Зачем в наше время нужно изучать язык ассемблера?
A: Если вы не разработчик ОС[8], вам наверное не нужно писать на ассемблере: современные компиляторы оптимизируют код намного лучше человека[9]. К тому же, современные CPU[10] это крайне сложные устройства и знание ассемблера вряд ли поможет узнать их внутренности. Но все-таки остается по крайней мере две области, где знание ассемблера может хорошо помочь: 1) исследование malware (*зловредов*) с целью анализа; 2) лучшее понимание вашего скомпилированного кода в процессе отладки. Таким образом, эта книга предназначена для тех, кто хочет скорее понимать ассемблер, нежели писать на нем, и вот почему здесь масса примеров, связанных с результатами работы компиляторов.

Q: Я кликнул на ссылку внутри PDF-документа, как теперь вернуться назад?
A: В Adobe Acrobat Reader нажмите сочетание Alt+LeftArrow.

Q: Я не могу понять, стоит ли мне заниматься reverse engineering-ом.
A: Наверное, среднее время для освоения сокращенной LITE-версии — 1-2 месяца.

Q: Могу ли я распечатать эту книгу? Использовать её для обучения?
A: Конечно, поэтому книга и лицензирована под лицензией Creative Commons. Кто-то может захотеть скомпилировать свою собственную версию книги, читайте здесь об этом.

Q: Я хочу перевести вашу книгу на другой язык.
A: Прочитайте мою заметку для переводчиков.

Q: Как можно найти работу reverse engineer-а?
A: На reddit, посвященному RE[11], время от времени бывают hiring thread (2013 Q3, 2014). Посмотрите там. В смежном субреддите «netsec» имеется похожий тред: 2014 Q2.

Q: Куда пойти учиться в Украине?
A: НТУУ «КПИ»: «Аналіз програмного коду та бінарних вразливостей»; факультативы.

Q: У меня есть вопрос...
A: Напишите мне его емейлом (dennis(a)yurichev.com).

8 Операционная Система
9 Очень хороший текст на эту тему: [Fog13]
10 Central processing unit
11 reddit.com/r/ReverseEngineering/

О переводе на корейский язык

В январе 2015, издательство Acorn в Южной Корее сделало много работы в переводе и издании моей книги (по состоянию на август 2014) на корейский язык.

Она теперь доступна на их сайте.

Переводил Byungho Min (twitter/tais9).

Обложку нарисовал мой хороший знакомый художник Андрей Нечаевский: facebook/andydinka.

Они также имеют права на издании книги на корейском языке.

Так что если вы хотите иметь *настоящую* книгу на полке на корейском языке и хотите поддержать мою работу, вы можете купить её.

Часть I

Образцы кода

Всё познается в сравнении

Автор неизвестен

Когда автор этой книги учил Си, а затем Си++, он просто писал небольшие фрагменты кода, компилировал и смотрел, что получилось на ассемблере. Так было намного проще понять[12]. Он делал это такое количество раз, что связь между кодом на Си/Си++ и тем, что генерирует компилятор, вбилась в его подсознание достаточно глубоко. После этого не трудно, глядя на код на ассемблере, сразу в общих чертах понимать, что там было написано на Си. Возможно это поможет кому-то ещё.

Иногда здесь используются достаточно древние компиляторы, чтобы получить самый короткий (или простой) фрагмент кода.

Уровни оптимизации и отладочная информация

Исходный код можно компилировать различными компиляторами с различными уровнями оптимизации. В типичном компиляторе этих уровней около трёх, где нулевой уровень — отключить оптимизацию. Различают также направления оптимизации кода по размеру и по скорости.

Неоптимизирующий компилятор работает быстрее, генерирует более понятный (хотя и более объемный) код. Оптимизирующий компилятор работает медленнее и старается сгенерировать более быстрый (хотя и не обязательно краткий) код.

Наряду с уровнями и направлениями оптимизации компилятор может включать в конечный файл отладочную информацию, производя таким образом код, который легче отлаживать.

Одна очень важная черта отладочного кода в том, что он может содержать связи между каждой строкой в исходном коде и адресом в машинном коде. Оптимизирующие компиляторы обычно генерируют код, где целые строки из исходного кода могут быть оптимизированы и не присутствовать в итоговом машинном коде.

Практикующий reverse engineer обычно сталкивается с обоими версиями, потому что некоторые разработчики включают оптимизацию, некоторые другие — нет. Вот почему мы постараемся поработать с примерами для обоих версий.

[12]Честно говоря, он и до сих пор так делаю, когда не понимают, как работает некий код.

Глава 1

Краткое введение в CPU

CPU это устройство исполняющее все программы.

Немного терминологии:

Инструкция : примитивная команда CPU. Простейшие примеры: перемещение между регистрами, работа с памятью, примитивные арифметические операции . Как правило, каждый CPU имеет свой набор инструкций (ISA[1]).

Машинный код : код понимаемый CPU. Каждая инструкция обычно кодируется несколькими байтами.

Язык ассемблера : машинный код плюс некоторые расширения, призванные облегчить труд программиста: макросы, имена, и т.д.

Регистр CPU : Каждый CPU имеет некоторый фиксированный набор регистров общего назначения (GPR[2]). ≈ 8 в x86, ≈ 16 в x86-64, ≈ 16 в ARM. Проще всего понимать регистр как временную переменную без типа . Можно представить, что вы пишете на ЯП[3] высокого уровня и у вас только 8 переменных шириной 32 (или 64) бита . Можно сделать очень много используя только их!

Откуда взялась разница между машинным кодом и ЯП высокого уровня? Ответ в том, что люди и CPU-ы отличаются друг от друга — . Человеку проще писать на ЯП высокого уровня вроде Си/Си++, Java, Python, а CPU проще работать с абстракциями куда более низкого уровня . Возможно, можно было бы придумать CPU исполняющий код ЯП высокого уровня, но он был бы значительно сложнее, чем те, что мы имеем сегодня . И наоборот, человеку очень неудобно писать на ассемблере из-за его низкоуровневости, к тому же, крайне трудно обойтись без мелких ошибок. Программа, переводящая код из ЯП высокого уровня в ассемблер называется *компилятором*[4].

[1] Instruction Set Architecture (Архитектура набора команд)
[2] General Purpose Registers (регистры общего пользования)
[3] Язык Программирования
[4] В более старой русскоязычной литературе также часто встречается термин «транслятор».

Глава 2

Простейшая функция

Наверное, простейшая из возможных функций это та что возвращает некоторую константу:

Вот, например:

Листинг 2.1: Код на Си/Си++

```
int f()
{
        return 123;
};
```

Скомпилируем её!

2.1. x86

И вот что делает оптимизирующий GCC:

Листинг 2.2: Оптимизирующий GCC/MSVC (вывод на ассемблере)

```
f:
        mov     eax, 123
        ret
```

Здесь только две инструкции. Первая помещает значение 123 в регистр EAX, который используется для передачи возвращаемых значений. Вторая это RET, которая возвращает управление в вызывающую функцию. Вызывающая функция возьмет результат из регистра EAX.

Нужно отметить, что название инструкции MOV в x86 и ARM сбивает с толку. На самом деле, данные не *перемещаются*, а скорее *копируются*.

Глава 3

Hello, world!

Продолжим, используя знаменитый пример из книги "The C programming Language"[Ker88]:

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

3.1. x86

3.1.1. MSVC

Компилируем в MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(Ключ /Fa означает сгенерировать листинг на ассемблере)

Листинг 3.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

Компилятор сгенерировал файл `1.obj`, который впоследствии будет слинкован линкером в `1.exe`. В нашем случае этот файл состоит из двух сегментов: CONST (для данных-констант) и _TEXT (для кода).

Строка `hello, world` в Си/Си++ имеет тип `const char[]` [Str13, p176, 7.3.2], однако не имеет имени. Но компилятору нужно как-то с ней работать, поэтому он дает ей внутреннее имя $SG3830.

Поэтому пример можно было бы переписать вот так:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Вернемся к листингу на ассемблере. Как видно, строка заканчивается нулевым байтом — это требования стандарта Си/Си++ для строк. Больше о строках в Си: 25.1.1 (стр. 112).

В сегменте кода _TEXT находится пока только одна функция: main(). Функция main(), как и практически все функции, начинается с пролога и заканчивается эпилогом[1].

Далее следует вызов функции printf() : CALL _printf. Перед этим вызовом адрес строки (или указатель на неё) с нашим приветствием при помощи инструкции PUSH помещается в стек.

После того, как функция printf() возвращает управление в функцию main(), адрес строки (или указатель на неё) всё ещё лежит в стеке. Так как он больше не нужен, то указатель стека (регистр ESP) корректируется.

ADD ESP, 4 означает прибавить 4 к значению в регистре ESP. Почему 4? Так как это 32-битный код, для передачи адреса нужно 4 байта. В x64-коде это 8 байт. ADD ESP, 4 эквивалентно POP регистр, но без использования какого-либо регистра[2].

Некоторые компиляторы, например, Intel C++ Compiler, в этой же ситуации могут вместо ADD сгенерировать POP ECX (подобное можно встретить, например, в коде Oracle RDBMS, им скомпилированном), что почти то же самое, только портится значение в регистре ECX. Возможно, компилятор применяет POP ECX, потому что эта инструкция короче (1 байт у POP против 3 у ADD).

Вот пример использования POP вместо ADD из Oracle RDBMS:

Листинг 3.2: Oracle RDBMS 10.2 Linux (файл app.o)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

После вызова printf() в оригинальном коде на Си/Си++ указано return 0 — вернуть 0 в качестве результата функции main(). В сгенерированном коде это обеспечивается инструкцией XOR EAX, EAX. XOR, как легко догадаться — «исключающее ИЛИ»[3], но компиляторы часто используют его вместо простого MOV EAX, 0 — снова потому, что опкод короче (2 байта у XOR против 5 у MOV).

Некоторые компиляторы генерируют SUB EAX, EAX, что значит *отнять значение в* EAX *от значения в* EAX, что в любом случае даст 0 в результате.

Самая последняя инструкция RET возвращает управление в вызывающую функцию. Обычно это код Си/Си++ CRT[4], который, в свою очередь, вернёт управление операционной системе.

3.2. x86-64

3.2.1. MSVC — x86-64

Попробуем также 64-битный MSVC:

Листинг 3.3: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
```

[1] Об этом смотрите подробнее в разделе о прологе и эпилоге функции (4 (стр. 8)).
[2] Флаги процессора, впрочем, модифицируются
[3] wikipedia
[4] C runtime library

```
        add     rsp, 40
        ret     0
main    ENDP
```

В x86-64 все регистры были расширены до 64-х бит и теперь имеют префикс R-. Чтобы поменьше задействовать стек (иными словами, поменьше обращаться кэшу и внешней памяти), уже давно имелся довольно популярный метод передачи аргументов функции через регистры (fastcall). Т.е. часть аргументов функции передается через регистры и часть — через стек. В Win64 первые 4 аргумента функции передаются через регистры RCX, RDX, R8, R9. Это мы здесь и видим: указатель на строку в `printf()` теперь передается не через стек, а через регистр RCX.

Указатели теперь 64-битные, так что они передаются через 64-битные части регистров (имеющие префикс R-). Но для обратной совместимости можно обращаться и к нижним 32 битам регистров используя префикс E-.

Вот как выглядит регистр RAX/EAX/AX/AL в x86-64:

<table>
<tr><td>7 (номер байта)</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

Функция `main()` возвращает значение типа *int*, который в Си/Си++, вероятно для лучшей совместимости и переносимости, оставили 32-битным. Вот почему в конце функции `main()` обнуляется не RAX, а EAX, т.е. 32-битная часть регистра.

Также видно, что 40 байт выделяются в локальном стеке. Это «shadow space», которое мы будем рассматривать позже: 8.2.1 (стр. 29).

3.3. Вывод

Основная разница между кодом x86/ARM и x64/ARM64 в том, что указатель на строку теперь 64-битный. Действительно, ведь для того современные CPU и стали 64-битными, потому что подешевела память, её теперь можно поставить в компьютер намного больше, и чтобы её адресовать, 32-х бит уже недостаточно. Поэтому все указатели теперь 64-битные.

Глава 4

Пролог и эпилог функций

Пролог функции это инструкции в самом начале функции. Как правило это что-то вроде такого фрагмента кода:

```
    push    ebp
    mov     ebp, esp
    sub     esp, X
```

Эти инструкции делают следующее: сохраняют значение регистра EBP на будущее, выставляют EBP равным ESP, затем подготавливают место в стеке для хранения локальных переменных.

EBP сохраняет свое значение на протяжении всей функции, он будет использоваться здесь для доступа к локальным переменным и аргументам. Можно было бы использовать и ESP, но он постоянно меняется и это не очень удобно.

Эпилог функции аннулирует выделенное место в стеке, восстанавливает значение EBP на старое и возвращает управление в вызывающую функцию:

```
    mov    esp, ebp
    pop    ebp
    ret    0
```

Пролог и эпилог функции обычно находятся в дизассемблерах для отделения функций друг от друга.

4.1. Рекурсия

Наличие эпилога и пролога может несколько ухудшить эффективность рекурсии.

Больше о рекурсии в этой книге: **??** (стр. **??**).

Глава 5

Стек

Стек в информатике — это одна из наиболее фундаментальных структур данных[1].

Технически это просто блок памяти в памяти процесса + регистр ESP в x86 или RSP в x64, либо SP[2] в ARM, который указывает где-то в пределах этого блока.

Часто используемые инструкции для работы со стеком — это PUSH и POP (в x86 и Thumb-режиме ARM). PUSH уменьшает ESP/RSP/SP на 4 в 32-битном режиме (или на 8 в 64-битном), затем записывает по адресу, на который указывает ESP/RSP/SP, содержимое своего единственного операнда.

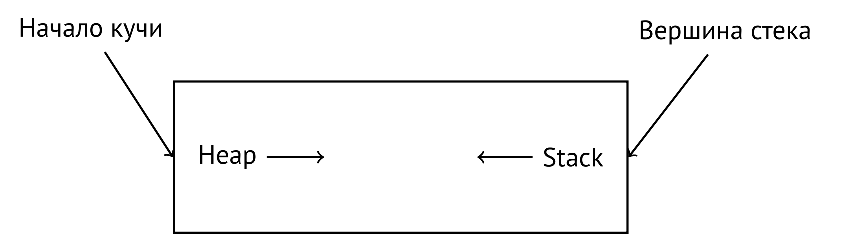
POP это обратная операция — сначала достает из указателя стека значение и помещает его в операнд (который очень часто является регистром) и затем увеличивает указатель стека на 4 (или 8).

В самом начале регистр-указатель указывает на конец стека. PUSH уменьшает регистр-указатель, а POP — увеличивает. Конец стека находится в начале блока памяти, выделенного под стек. Это странно, но это так.

5.1. Почему стек растет в обратную сторону?

Интуитивно мы можем подумать, что, как и любая другая структура данных, стек мог бы расти вперед, т.е. в сторону увеличения адресов.

Причина, почему стек растет назад, вероятно, историческая. Когда компьютеры были большие и занимали целую комнату, было очень легко разделить сегмент на две части: для кучи и для стека. Заранее было неизвестно, насколько большой может быть куча или стек, так что это решение было самым простым.



В [RT74] можно прочитать:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

Это немного напоминает как некоторые студенты пишут два конспекта в одной тетрадке: первый конспект начинается обычным образом, второй пишется с конца, перевернув тетрадку. Конспекты могут встретиться где-то посредине, в случае недостатка свободного места.

[1] wikipedia.org/wiki/Call_stack
[2] stack pointer. SP/ESP/RSP в x86/x64. SP в ARM.

5.2. Для чего используется стек?

5.2.1. Сохранение адреса возврата управления

x86

При вызове другой функции через CALL сначала в стек записывается адрес, указывающий на место после инструкции CALL, затем делается безусловный переход (почти как JMP) на адрес, указанный в операнде.

CALL — это аналог пары инструкций PUSH `address_after_call / JMP`.

RET вытаскивает из стека значение и передает управление по этому адресу — это аналог пары инструкций POP `tmp / JMP tmp`.

Крайне легко устроить переполнение стека, запустив бесконечную рекурсию:

```
void f()
{
        f();
};
```

MSVC 2008 предупреждает о проблеме:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause runtime ↙
    ↘ stack overflow
```

...но, тем не менее, создает нужный код:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

...причем, если включить оптимизацию (/Ox), то будет даже интереснее, без переполнения стека, но работать будет *корректно*[3]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                          ; f
```

5.2.2. Передача параметров функции

Самый распространенный способ передачи параметров в x86 называется «cdecl»:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

[3] здесь ирония

Вызываемая функция получает свои параметры также через указатель стека.

Следовательно, так расположены значения в стеке перед исполнением самой первой инструкции функции `f()`:

ESP	адрес возврата
ESP+4	аргумент#1, маркируется в IDA[4] как `arg_0`
ESP+8	аргумент#2, маркируется в IDA как `arg_4`
ESP+0xC	аргумент#3, маркируется в IDA как `arg_8`
…	…

Важно отметить, что, в общем, никто не заставляет программистов передавать параметры именно через стек, это не является требованием к исполняемому коду. Вы можете делать это совершенно иначе, не используя стек вообще.

К примеру, можно выделять в куче место для аргументов, заполнять их и передавать в функцию указатель на это место через EAX. И это вполне будет работать[5]. Однако традиционно сложилось, что в x86 и ARM передача аргументов происходит именно через стек.

Кстати, вызываемая функция не имеет информации о количестве переданных ей аргументов. Функции Си с переменным количеством аргументов (как `printf()`) определяют их количество по спецификаторам строки формата (начинающиеся со знака %). Если написать что-то вроде

```
printf("%d %d %d", 1234);
```

`printf()` выведет 1234, затем ещё два случайных числа, которые волею случая оказались в стеке рядом.

Вот почему не так уж и важно, как объявлять функцию `main()` : как `main()`, `main(int argc, char *argv[])` либо `main(int argc, char *argv[], char *envp[])`.

В реальности, CRT-код вызывает `main()` примерно так:

```
push envp
push argv
push argc
call main
...
```

Если вы объявляете `main()` без аргументов, они, тем не менее, присутствуют в стеке, но не используются. Если вы объявите `main()` как `main(int argc, char *argv[])`, вы можете использовать два первых аргумента, а третий останется для вашей функции «невидимым». Более того, можно даже объявить `main(int argc)`, и это будет работать.

5.2.3. Хранение локальных переменных

Функция может выделить для себя некоторое место в стеке для локальных переменных, просто отодвинув указатель стека глубже к концу стека. Это очень быстро вне зависимости от количества локальных переменных.

Хранить локальные переменные в стеке не является необходимым требованием. Вы можете хранить локальные переменные где угодно. Но по традиции всё сложилось так.

5.2.4. x86: Функция alloca()

Интересен случай с функцией `alloca()`[6].

Эта функция работает как `malloc()`, но выделяет память прямо в стеке.

Память освобождать через `free()` не нужно, так как эпилог функции (4 (стр. 8)) вернет ESP в изначальное состояние и выделенная память просто *выкидывается*.

Интересна реализация функции `alloca()`.

Эта функция, если упрощенно, просто сдвигает ESP вглубь стека на столько байт, сколько вам нужно и возвращает ESP в качестве указателя на выделенный блок. Попробуем:

[5] Например, в книге Дональда Кнута «Искусство программирования», в разделе 1.4.1 посвященном подпрограммам [Knu98, раздел 1.4.1], мы можем прочитать о возможности располагать параметры для вызываемой подпрограммы после инструкции JMP, передающей управление подпрограмме. Кнут описывает, что это было особенно удобно для компьютеров IBM System/360.

[6] В MSVC, реализацию функции можно посмотреть в файлах `alloca16.asm` и `chkstk.asm` в `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Функция `_snprintf()` работает так же, как и `printf()`, только вместо выдачи результата в stdout (т.е. на терминал или в консоль), записывает его в буфер `buf`. Функция `puts()` выдает содержимое буфера `buf` в stdout. Конечно, можно было бы заменить оба этих вызова на один `printf()`, но здесь нужно проиллюстрировать использование небольшого буфера.

MSVC

Компилируем (MSVC 2010):

Листинг 5.1: MSVC 2010

```
...

    mov    eax, 600         ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp

    push   3
    push   2
    push   1
    push   OFFSET $SG2672
    push   600              ; 00000258H
    push   esi
    call   __snprintf

    push   esi
    call   _puts
    add    esp, 28          ; 0000001cH

...
```

Единственный параметр в `alloca()` передается через EAX, а не как обычно через стек[7]. После вызова `alloca()` ESP указывает на блок в 600 байт, который мы можем использовать под `buf`.

5.2.5. (Windows) SEH

В стеке хранятся записи SEH[10] для функции (если они присутствуют).

5.2.6. Защита от переполнений буфера

Здесь больше об этом (16.2 (стр. 63)).

[7]Это потому, что alloca() — это не сколько функция, сколько т.н. *compiler intrinsic*.
Одна из причин, почему здесь нужна именно функция, а не несколько инструкций прямо в коде в том, что в реализации функции alloca() от MSVC[8] есть также код, читающий из только что выделенной памяти, чтобы ОС подключила физическую память к этому региону VM[9].
[10]Structured Exception Handling

5.2.7. Автоматическое освобождение данных в стеке

Возможно, причина хранения локальных переменных и SEH-записей в стеке в том, что после выхода из функции, всё эти данные освобождаются автоматически, используя только одну инструкцию корректирования указателя стека (часто это ADD). Аргументы функций, можно сказать, тоже освобождаются автоматически в конце функции. А всё что хранится в куче (*heap*) нужно освобождать явно.

5.3. Разметка типичного стека

Разметка типичного стека в 32-битной среде перед исполнением самой первой инструкции функции выглядит так:

…	…
ESP-0xC	локальная переменная #2, маркируется в IDA как `var_8`
ESP-8	локальная переменная #1, маркируется в IDA как `var_4`
ESP-4	сохраненное значение EBP
ESP	адрес возврата
ESP+4	аргумент#1, маркируется в IDA как `arg_0`
ESP+8	аргумент#2, маркируется в IDA как `arg_4`
ESP+0xC	аргумент#3, маркируется в IDA как `arg_8`
…	…

Глава 6

printf() с несколькими аргументами

Попробуем теперь немного расширить пример *Hello, world!* (3 (стр. 5)), написав в теле функции main():

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

6.1. x86

6.1.1. x86: 3 аргумента

MSVC

Компилируем при помощи MSVC 2010 Express, и в итоге получим:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Всё почти то же, за исключением того, что теперь видно, что аргументы для printf() заталкиваются в стек в обратном порядке: самый первый аргумент заталкивается последним.

Кстати, вспомним, что переменные типа *int* в 32-битной системе, как известно, имеет ширину 32 бита, это 4 байта .

Итак, у нас всего 4 аргумента. $4 * 4 = 16$ — именно 16 байт занимают в стеке указатель на строку плюс ещё 3 числа типа *int*.

Когда при помощи инструкции ADD ESP, X корректируется указатель стека ESP после вызова какой-либо функции, зачастую можно сделать вывод о том, сколько аргументов у вызываемой функции было, разделив X на 4.

Конечно, это относится только к cdecl-методу передачи аргументов через стек, и только для 32-битной среды.

Иногда бывает так, что подряд идут несколько вызовов разных функций, но стек корректируется только один раз, после последнего вызова:

```
push a1
push a2
call ...
...
push a1
call ...
```

```
...
push a1
push a2
push a3
call ...
add esp, 24
```

Вот пример из реальной жизни:

Листинг 6.1: x86

```
.text:100113E7                push    3
.text:100113E9                call    sub_100018B0 ; берет один аргумент (3)
.text:100113EE                call    sub_100019D0 ; не имеет аргументов вообще
.text:100113F3                call    sub_10006A90 ; не имеет аргументов вообще
.text:100113F8                push    1
.text:100113FA                call    sub_100018B0 ; берет один аргумент (1)
.text:100113FF                add     esp, 8       ; выбрасывает из стека два аргумента
```

6.1.2. x64: 8 аргументов

Для того чтобы посмотреть, как остальные аргументы будут передаваться через стек, изменим пример ещё раз, увеличив количество передаваемых аргументов до 9 (строка формата `printf()` и 8 переменных типа *int*):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

Как уже было сказано ранее, первые 4 аргумента в Win64 передаются в регистрах RCX, RDX, R8, R9 , а остальные — через стек. Здесь мы это и видим. Впрочем, инструкция PUSH не используется, вместо неё при помощи MOV значения сразу записываются в стек.

Листинг 6.2: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; возврат 0
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Наблюдательный читатель может спросить, почему для значений типа *int* отводится 8 байт, ведь нужно только 4? Да, это нужно запомнить: для значений всех типов более коротких чем 64-бита, отводится 8 байт. Это сделано для удобства: так

всегда легко рассчитать адрес того или иного аргумента. К тому же, все они расположены по выровненным адресам в памяти. В 32-битных средах точно также: для всех типов резервируется 4 байта в стеке.

6.2. Вывод

Вот примерный скелет вызова функции:

Листинг 6.3: x86

```
...
PUSH третий аргумент
PUSH второй аргумент
PUSH первый аргумент
CALL функция
; модифицировать указатель стека (если нужно)
```

Листинг 6.4: x64 (MSVC)

```
MOV RCX, первый аргумент
MOV RDX, второй аргумент
MOV R8, третий аргумент
MOV R9, 4-й аргумент
...
PUSH 5-й, 6-й аргумент, и т.д. (если нужно)
CALL функция
; модифицировать указатель стека (если нужно)
```

6.3. Кстати

Кстати, разница между способом передачи параметров принятая в x86, x64, fastcall, ARM и MIPS неплохо иллюстрирует тот важный момент, что процессору, в общем, всё равно, как будут передаваться параметры функций. Можно создать гипотетический компилятор, который будет передавать их при помощи указателя на структуру с параметрами, не пользуясь стеком вообще.

CPU не знает о соглашениях о вызовах вообще.

Можно также вспомнить, что начинающие программисты на ассемблере передают параметры в другие функции обычно через регистры, без всякого явного порядка, или даже через глобальные переменные. И всё это нормально работает.

Глава 7

scanf()

Теперь попробуем использовать scanf().

7.1. Простой пример

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Использовать `scanf()` в наши времена для того, чтобы спросить у пользователя что-то — не самая хорошая идея. Но так мы проиллюстрируем передачу указателя на переменную типа *int*.

7.1.1. Об указателях

Это одна из фундаментальных вещей в информатике. Часто большой массив, структуру или объект передавать в другую функцию путем копирования данных невыгодно, а передать адрес массива, структуры или объекта куда проще. К тому же, если вызываемая функция (callee) должна изменить что-то в этом большом массиве или структуре, то возвращать её полностью так же абсурдно. Так что самое простое, что можно сделать, это передать в функцию-callee адрес массива или структуры, и пусть callee что-то там изменит.

Указатель в Си/Си++ — это просто адрес какого-либо места в памяти.

В x86 адрес представляется в виде 32-битного числа (т.е. занимает 4 байта), а в x86-64 как 64-битное число (занимает 8 байт). Кстати, отсюда негодование некоторых людей, связанное с переходом на x86-64 — на этой архитектуре все указатели занимают в 2 раза больше места, в том числе и в “дорогой” кэш-памяти.

При некотором упорстве можно работать только с безтиповыми указателями (`void*`); например, стандартная функция Си `memcpy()`, копирующая блок из одного места памяти в другое, принимает на вход 2 указателя типа `void*`, потому что нельзя заранее предугадать, какого типа блок вы собираетесь копировать. Для копирования тип данных не важен, важен только размер блока.

Также указатели широко используются, когда функции нужно вернуть более одного значения (мы ещё вернемся к этому в будущем). Функция *scanf()* — это как раз такой случай. Помимо того, что этой функции нужно показать, сколько значений было прочитано успешно, ей ещё и нужно вернуть сами значения.

Тип указателя в Си/Си++ нужен для проверки типов на стадии компиляции. Внутри, в скомпилированном коде, никакой информации о типах указателей нет вообще.

7.1.2. x86

MSVC

Что получаем на ассемблере, компилируя в MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; возврат 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Переменная x является локальной.

По стандарту Си/Си++ она доступна только из этой же функции и нигде более. Так получилось, что локальные переменные располагаются в стеке. Может быть, можно было бы использовать и другие варианты, но в x86 это традиционно так.

Следующая после пролога инструкция PUSH ECX не ставит своей целью сохранить значение регистра ECX. (Заметьте отсутствие соответствующей инструкции POP ECX в конце функции).

Она на самом деле выделяет в стеке 4 байта для хранения x в будущем.

Доступ к x будет осуществляться при помощи объявленного макроса _x$ (он равен -4) и регистра EBP указывающего на текущий фрейм.

Во всё время исполнения функции EBP указывает на текущий фрейм и через EBP+смещение можно получить доступ как к локальным переменным функции, так и аргументам функции.

Можно было бы использовать ESP, но он во время исполнения функции часто меняется, а это не удобно. Так что можно сказать, что EBP это *замороженное состояние* ESP на момент начала исполнения функции.

Разметка типичного стекового фрейма в 32-битной среде:

...	...
EBP-8	локальная переменная #2, маркируется в IDA как var_8
EBP-4	локальная переменная #1, маркируется в IDA как var_4
EBP	сохраненное значение EBP
EBP+4	адрес возврата
EBP+8	аргумент#1, маркируется в IDA как arg_0
EBP+0xC	аргумент#2, маркируется в IDA как arg_4
EBP+0x10	аргумент#3, маркируется в IDA как arg_8
...	...

У функции `scanf()` в нашем примере два аргумента.

Первый — указатель на строку, содержащую %d и второй — адрес переменной x.

Вначале адрес x помещается в регистр EAX при помощи инструкции `lea eax, DWORD PTR _x$[ebp]`.

Можно сказать, что в данном случае LEA просто помещает в EAX результат суммы значения в регистре EBP и макроса `_x$`.

Это тоже что и `lea eax, [ebp-4]`.

Итак, от значения EBP отнимается 4 и помещается в EAX. Далее значение EAX заталкивается в стек и вызывается `scanf()`.

После этого вызывается `printf()`. Первый аргумент вызова строка: `You entered %d...\n`.

Второй аргумент: `mov ecx, [ebp-4]`. Эта инструкция помещает в ECX не адрес переменной x, а её значение.

Далее значение ECX заталкивается в стек и вызывается `printf()`.

Кстати

Кстати, этот простой пример иллюстрирует то обстоятельство, что компилятор преобразует список выражений в Си/Си++-блоке просто в последовательный набор инструкций. Между выражениями в Си/Си++ ничего нет, и в итоговом машинном коде между ними тоже ничего нет, управление переходит от одной инструкции к следующей за ней.

7.1.3. x64

Всё то же самое, только используются регистры вместо стека для передачи аргументов функций.

MSVC

Листинг 7.1: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; возврат 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
```

```
_TEXT    ENDS
```

7.2. Глобальные переменные

А что если переменная x из предыдущего примера будет глобальной переменной, а не локальной? Тогда к ней смогут обращаться из любого другого места, а не только из тела функции. Глобальные переменные считаются анти-паттерном, но ради примера мы можем себе это позволить.

```
#include <stdio.h>

// теперь x это глобальная переменная
int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

7.2.1. MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

В целом ничего особенного. Теперь x объявлена в сегменте _DATA. Память для неё в стеке более не выделяется. Все обращения к ней происходит не через стек, а уже напрямую. Неинициализированные глобальные переменные не занимают места в исполняемом файле (и действительно, зачем в исполняемом файле нужно выделять место под изначально нулевые переменные?), но тогда, когда к этому месту в памяти кто-то обратится, ОС подставит туда блок, состоящий из нулей[1].

[1] Так работает VM

Попробуем изменить объявление этой переменной:

```
int x=10; // значение по умолчанию
```

Выйдет в итоге:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Здесь уже по месту этой переменной записано 0xA с типом DD (dword = 32 бита).

Если вы откроете скомпилированный .exe-файл в IDA, то увидите, что *x* находится в начале сегмента _DATA, после этой переменной будут текстовые строки.

А вот если вы откроете в IDA.exe скомпилированный в прошлом примере, где значение *x* не определено, то вы увидите:

```
.data:0040FA80 _x               dd ?                    ; DATA XREF: _main+10
.data:0040FA80                                          ; _main+22
.data:0040FA84 dword_40FA84     dd ?                    ; DATA XREF: _memset+1E
.data:0040FA84                                          ; unknown_libname_1+28
.data:0040FA88 dword_40FA88     dd ?                    ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                                          ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem            dd ?                    ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                                          ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90     dd ?                    ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                                          ; __calloc_impl+72
.data:0040FA94 dword_40FA94     dd ?                    ; DATA XREF: ___sbh_free_block+2FE
```

_x обозначен как ?, наряду с другими переменными не требующими инициализации. Это означает, что при загрузке .exe в память, место под всё это выделено будет и будет заполнено нулевыми байтами [ISO07, 6.7.8p10]. Но в самом .exe ничего этого нет. Неинициализированные переменные не занимают места в исполняемых файлах. Это удобно для больших массивов, например.

7.2.2. MSVC: x64

Листинг 7.2: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; возврат 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Почти такой же код как и в x86. Обратите внимание что для `scanf()` адрес переменной x передается при помощи инструкции LEA, а во второй `printf()` передается само значение переменной при помощи MOV . DWORD PTR — это часть языка ассемблера (не имеющая отношения к машинным кодам) показывающая, что тип переменной в памяти именно 32-битный, и инструкция MOV должна быть здесь закодирована соответственно.

7.3. Проверка результата scanf()

Как уже было упомянуто, использовать `scanf()` в наше время слегка старомодно. Но если уж жизнь заставила этим заниматься, нужно хотя бы проверять, сработал ли `scanf()` правильно или пользователь ввел вместо числа что-то другое, что `scanf()` не смог трактовать как число.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

По стандарту, `scanf()`[2] возвращает количество успешно полученных значений.

В нашем случае, если всё успешно и пользователь ввел таки некое число, `scanf()` вернет 1. А если нет, то 0 (или EOF[3]).

Добавим код, проверяющий результат `scanf()` и в случае ошибки он сообщает пользователю что-то другое.

Это работает предсказуемо:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

7.3.1. MSVC: x86

Вот что выходит на ассемблере (MSVC 2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
```

[2] scanf, wscanf: MSDN
[3] End of file (конец файла)

```
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

Для того чтобы вызывающая функция имела доступ к результату вызываемой функции, вызываемая функция (в нашем случае `scanf()`) оставляет это значение в регистре EAX.

Мы проверяем его инструкцией `CMP EAX, 1` (*CoMPare*), то есть сравниваем значение в EAX с 1.

Следующий за инструкцией CMP: условный переход JNE. Это означает *Jump if Not Equal*, то есть условный переход *если не равно*.

Итак, если EAX не равен 1, то JNE заставит CPU перейти по адресу указанном в операнде JNE, у нас это `$LN2@main`. Передав управление по этому адресу, CPU начнет исполнять вызов `printf()` с аргументом `What you entered? Huh?`. Но если всё нормально, перехода не случится и исполнится другой `printf()` с двумя аргументами: `'You entered %d...'` и значением переменной x.

Для того чтобы после этого вызова не исполнился сразу второй вызов `printf()`, после него есть инструкция JMP, безусловный переход, который отправит процессор на место после второго `printf()` и перед инструкцией `XOR EAX, EAX`, которая реализует `return 0`.

Итак, можно сказать что в подавляющих случаях сравнение какой-либо переменной с чем-то другим происходит при помощи пары инструкций CMP и `Jcc`, где *cc* это *condition code*. CMP сравнивает два значения и выставляет флаги процессора[4]. `Jcc` проверяет нужные ему флаги и выполняет переход по указанному адресу (или не выполняет).

Но на самом деле, как это не парадоксально поначалу звучит, CMP это почти то же самое что и инструкция SUB, которая отнимает числа одно от другого. Все арифметические инструкции также выставляют флаги в соответствии с результатом, не только CMP. Если мы сравним 1 и 1, от единицы отнимется единица, получится 0, и выставится флаг ZF (*zero flag*), означающий, что последний полученный результат был 0. Ни при каких других значениях EAX, флаг ZF не может быть выставлен, кроме тех, когда операнды равны друг другу. Инструкция JNE проверяет только флаг ZF, и совершает переход только если флаг не поднят. Фактически, JNE это синоним инструкции JNZ (*Jump if Not Zero*). Ассемблер транслирует обе инструкции в один и тот же опкод. Таким образом, можно CMP заменить на SUB и всё будет работать также, но разница в том, что SUB всё-таки испортит значение в первом операнде. CMP это *SUB без сохранения результата, но изменяющая флаги*.

[4] См. также о флагах x86-процессора: wikipedia.

7.3.2. MSVC: x86 + Hiew

Это ещё может быть и простым примером исправления исполняемого файла. Мы можем попробовать исправить его таким образом, что программа всегда будет выводить числа, вне зависимости от ввода.

Исполняемый файл скомпилирован с импортированием функций из `MSVCR*.DLL` (т.е. с опцией `/MD`)[5], поэтому мы можем отыскать функцию `main()` в самом начале секции `.text`. Откроем исполняемый файл в Hiew, найдем самое начало секции `.text` (Enter, F8, F6, Enter, Enter).

Мы увидим следующее:

```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe           □FRO --------        a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                           push        ebp
.00401001: 8BEC                         mov         ebp,esp
.00401003: 51                           push        ecx
.00401004: 6800304000                   push        000403000 ;'Enter X:' --□1
.00401009: FF1594204000                 call        printf
.0040100F: 83C404                       add         esp,4
.00401012: 8D45FC                       lea         eax,[ebp][-4]
.00401015: 50                           push        eax
.00401016: 680C304000                   push        00040300C --□2
.0040101B: FF158C204000                 call        scanf
.00401021: 83C408                       add         esp,8
.00401024: 83F801                       cmp         eax,1
.00401027: 7514                         jnz        .00040103D --□3
.00401029: 8B4DFC                       mov         ecx,[ebp][-4]
.0040102C: 51                           push        ecx
.0040102D: 6810304000                   push        000403010 ;'You entered %d...' --□4
.00401032: FF1594204000                 call        printf
.00401038: 83C408                       add         esp,8
.0040103B: EB0E                         jmps       .00040104B --□5
.0040103D: 6824304000                  3push        000403024 ;'What you entered? Huh?' --□6
.00401042: FF1594204000                 call        printf
.00401048: 83C404                       add         esp,4
.0040104B: 33C0                        5xor         eax,eax
.0040104D: 8BE5                         mov         esp,ebp
.0040104F: 5D                           pop         ebp
.00401050: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                   mov         eax,000005A4D ;'  ZM'
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam
```

Рис. 7.1: Hiew: функция `main()`

Hiew находит ASCIIZ[6]-строки и показывает их, также как и имена импортируемых функций.

[5] то, что ещё называют «dynamic linking»

[6] ASCII Zero (ASCII-строка заканчивающаяся нулем)

Переведите курсор на адрес .00401027 (с инструкцией JNZ, которую мы хотим заблокировать), нажмите F3, затем наберите «9090»(что означает два NOP[7]-а):

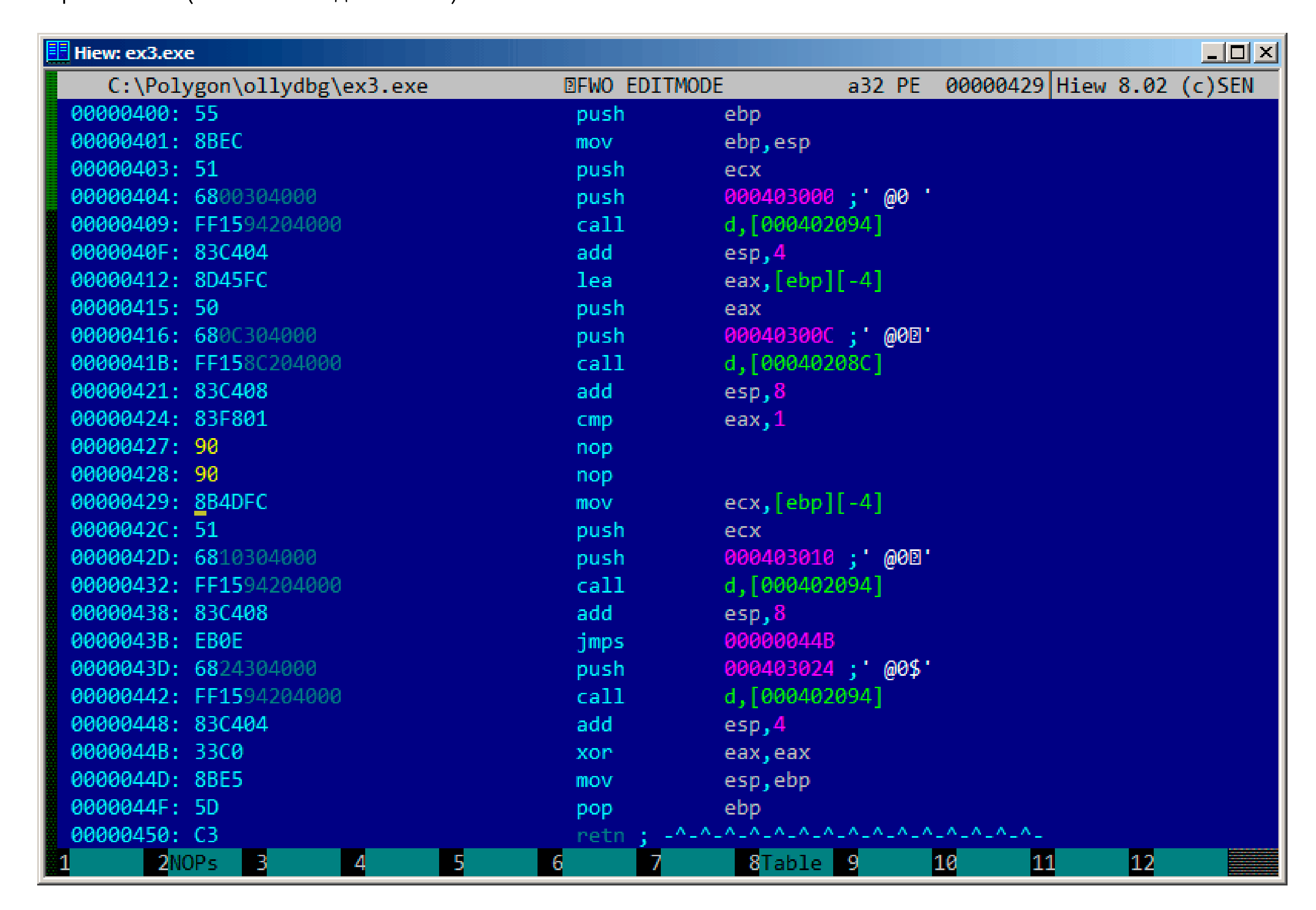


Рис. 7.2: Hiew: замена JNZ на два NOP-а

Затем F9 (update). Теперь исполняемый файл записан на диск. Он будет вести себя так, как нам надо.

Два NOP-а возможно, не так эстетично, как могло бы быть. Другой способ изменить инструкцию это записать 0 во второй байт опкода (смещение перехода), так что JNZ всегда будет переходить на следующую инструкцию.

Можно изменить и наоборот: первый байт заменить на EB, второй байт (смещение перехода) не трогать. Получится всегда срабатывающий безусловный переход. Теперь сообщение об ошибке будет выдаваться всегда, даже если мы ввели число.

7.3.3. MSVC: x64

Так как здесь мы работаем с переменными типа *int*, а они в x86-64 остались 32-битными, то мы здесь видим, как продолжают использоваться регистры с префиксом E-. Но для работы с указателями, конечно, используются 64-битные части регистров с префиксом R-.

Листинг 7.3: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
```

[7] No OPeration

```
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; возврат 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

7.4. Упражнения

7.4.1. Упражнение #1

Этот код, когда компилируется при помощи GCC в Linux x86-64, падает во время исполнения (segmentation fault). Но он работает в среде Windows, когда скомпилирован при помощи MSVC 2010 x86. Почему?

```
#include <string.h>
#include <stdio.h>

void alter_string(char *s)
{
        strcpy (s, "Goodbye!");
        printf ("Result: %s\n", s);
};

int main()
{
        alter_string ("Hello, world!\n");
};
```

Ответ: **??** (стр. **??**).

Глава 8

Доступ к переданным аргументам

Как мы уже успели заметить, вызывающая функция передает аргументы для вызываемой через стек. А как вызываемая функция получает к ним доступ?

Листинг 8.1: простой пример

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

8.1. x86

8.1.1. MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010 Express):

Листинг 8.2: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_c$ = 16                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; третий аргумент
        push    2 ; второй аргумент
        push    1 ; первый аргумент
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
```

```
        call    _printf
        add     esp, 8
        ; возврат 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Итак, здесь видно: в функции `main()` заталкиваются три числа в стек и вызывается функция `f(int,int,int)`. Внутри `f()` доступ к аргументам, также как и к локальным переменным, происходит через макросы: `_a$ = 8`, но разница в том, что эти смещения со знаком *плюс*, таким образом если прибавить макрос `_a$` к указателю на EBP, то адресуется *внешняя* часть фрейма стека относительно EBP.

Далее всё более-менее просто: значение a помещается в EAX. Далее EAX умножается при помощи инструкции `IMUL` на то, что лежит в _b, и в EAX остается произведение этих двух значений. Далее к регистру EAX прибавляется то, что лежит в _c. Значение из EAX никуда не нужно перекладывать, оно уже лежит где надо. Возвращаем управление вызываемой функции — она возьмет значение из EAX и отправит его в `printf()`.

8.2. x64

В x86-64 всё немного иначе, здесь аргументы функции (4 или 6) передаются через регистры, а callee из читает их из регистров, а не из стека.

8.2.1. MSVC

Оптимизирующий MSVC:

Листинг 8.3: Оптимизирующий MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - первый аргумент
        ; EDX - второй аргумент
        ; R8D - третий аргумент
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Как видно, очень компактная функция `f()` берет аргументы прямо из регистров. Инструкция LEA используется здесь для сложения чисел. Должно быть компилятор посчитал, что это будет эффективнее использования ADD. В самой `main()` LEA также используется для подготовки первого и третьего аргумента: должно быть, компилятор решил, что LEA будет работать здесь быстрее, чем загрузка значения в регистр при помощи MOV.

Попробуем посмотреть вывод неоптимизирующего MSVC:

Листинг 8.4: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
```

```
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - первый аргумент
                ; EDX - второй аргумент
                ; R8D - третий аргумент
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; третий аргумент
                mov     edx, 2 ; второй аргумент
                mov     ecx, 1 ; первый аргумент
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; возврат 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

Немного путаннее: все 3 аргумента из регистров зачем-то сохраняются в стеке. Это называется «shadow space» [1]: каждая функция в Win64 может (хотя и не обязана) сохранять значения 4-х регистров там. Это делается по крайней мере из-за двух причин: 1) в большой функции отвести целый регистр (а тем более 4 регистра) для входного аргумента слишком расточительно, так что к нему будет обращение через стек; 2) отладчик всегда знает, где найти аргументы функции в момент останова[2].

Так что, какие-то большие функции могут сохранять входные аргументы в «shadows space» для использования в будущем, а небольшие функции, как наша, могут этого и не делать.

Место в стеке для «shadow space» выделяет именно caller.

[1] MSDN
[2] MSDN

Глава 9

Ещё о возвращаемых результатах

Результат выполнения функции в x86 обычно возвращается[1] через регистр EAX, а если результат имеет тип байт или символ (*char*), то в самой младшей части EAX — AL. Если функция возвращает число с плавающей запятой, то будет использован регистр FPU ST(0).

9.1. Попытка использовать результат функции возвращающей *void*

Кстати, что будет, если возвращаемое значение в функции main() объявлять не как *int*, а как *void*?

Т.н. startup-код вызывает main() примерно так:

```
push envp
push argv
push argc
call main
push eax
call exit
```

Иными словами:

```
exit(main(argc,argv,envp));
```

Если вы объявите main() как *void*, и ничего не будете возвращать явно (при помощи выражения *return*), то в единственный аргумент exit() попадет то, что лежало в регистре EAX на момент выхода из main(). Там, скорее всего, будет какие-то случайное число, оставшееся от работы вашей функции. Так что код завершения программы будет псевдослучайным.

Мы можем это проиллюстрировать. Заметьте, что у функции main() тип возвращаемого значения именно *void*:

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Скомпилируем в Linux.

GCC 4.8.1 заменила printf() на puts(), но это нормально, потому что puts() возвращает количество выведенных символов, так же как и printf(). Обратите внимание на то, что EAX не обнуляется перед выходом их main(). Это значит что EAX перед выходом из main() содержит то, что puts() оставляет там.

Листинг 9.1: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
```

[1] См. также: MSDN: Return Values (C++): MSDN

```
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Напишем небольшой скрипт на bash, показывающий статус возврата («exit status» или «exit code») :

Листинг 9.2: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

И запустим:

```
$ tst.sh
Hello, world!
14
```

14 это как раз количество выведенных символов.

9.2. Что если не использовать результат функции?

`printf()` возвращает количество успешно выведенных символов, но результат работы этой функции редко используется на практике. Можно даже явно вызывать функции, чей смысл именно в возвращаемых значениях, но явно не использовать их:

```
int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};
```

Результат работы rand() остается в EAX во всех четырех случаях. Но в первых трех случаях значение, лежащее в EAX, просто выбрасывается.

Глава 10

Оператор GOTO

Оператор GOTO считается анти-паттерном [Dij68], но тем не менее, его можно использовать в разумных пределах [Knu74], [Yur13, с. 1.3.2].

Вот простейший пример:

```
#include <stdio.h>

int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

Вот что мы получаем в MSVC 2012:

Листинг 10.1: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Выражение *goto* заменяется инструкцией JMP, которая работает точно также: безусловный переход в другое место.

Вызов второго `printf()` может исполнится только при помощи человеческого вмешательства, используя отладчик или модифицирование кода.

10.1. Мертвый код

Вызов второго `printf()` также называется «мертвым кодом» («dead code») в терминах компиляторов. Это значит, что он никогда не будет исполнен. Так что если вы компилируете этот пример с оптимизацией, компилятор удаляет «мертвый код» не оставляя следа:

Листинг 10.2: Оптимизирующий MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

Впрочем, строку «skip me!» компилятор убрать забыл.

Глава 11

Условные переходы

11.1. Простой пример

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

11.1.1. x86

x86 + MSVC

Имеем в итоге функцию f_signed():

Листинг 11.1: Неоптимизирующий MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
```

```
$LN3@f_signed:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_signed
    push    OFFSET $SG739        ; 'a==b'
    call    _printf
    add     esp, 4
$LN2@f_signed:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jge     SHORT $LN4@f_signed
    push    OFFSET $SG741        ; 'a<b'
    call    _printf
    add     esp, 4
$LN4@f_signed:
    pop     ebp
    ret     0
_f_signed ENDP
```

Первая инструкция JLE значит *Jump if Less or Equal*. Если второй операнд больше первого или равен ему, произойдет переход туда, где будет следующая проверка. А если это условие не срабатывает (то есть второй операнд меньше первого), то перехода не будет, и сработает первый `printf()`. Вторая проверка это JNE: *Jump if Not Equal*. Переход не произойдет, если операнды равны.

Третья проверка JGE: *Jump if Greater or Equal* — переход если первый операнд больше второго или равен ему. Кстати, если все три условных перехода сработают, ни один `printf()` не вызовется. Но без внешнего вмешательства это невозможно.

Функция `f_unsigned()` точно такая же, за тем исключением, что используются инструкции JBE и JAE вместо JLE и JGE:

Листинг 11.2: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jbe     SHORT $LN3@f_unsigned
    push    OFFSET $SG2761    ; 'a>b'
    call    _printf
    add     esp, 4
$LN3@f_unsigned:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_unsigned
    push    OFFSET $SG2763    ; 'a==b'
    call    _printf
    add     esp, 4
$LN2@f_unsigned:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jae     SHORT $LN4@f_unsigned
    push    OFFSET $SG2765    ; 'a<b'
    call    _printf
    add     esp, 4
$LN4@f_unsigned:
    pop     ebp
    ret     0
_f_unsigned ENDP
```

Здесь всё то же самое, только инструкции условных переходов немного другие: JBE — *Jump if Below or Equal* и JAE — *Jump if Above or Equal*. Эти инструкции (JA/JAE/JB/JBE) отличаются от JG/JGE/JL/JLE тем, что работают с беззнаковыми переменными.

Отступление: смотрите также секцию о представлении знака в числах (22 (стр. 105)). Таким образом, увидев где используется JG/JL вместо JA/JB и наоборот, можно сказать почти уверенно насчет того, является ли тип переменной знаковым (signed) или беззнаковым (unsigned).

Далее функция `main()`, где ничего нового для нас нет:

Листинг 11.3: main()

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + Hiew

Можем попробовать модифицировать исполняемый файл так, чтобы функция `f_unsigned()` всегда показывала «a==b», при любых входящих значениях. Вот как она выглядит в Hiew:

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe          ↓FRO --------        a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                          push         ebp
.00401001: 8BEC                        mov          ebp,esp
.00401003: 8B4508                      mov          eax,[ebp][8]
.00401006: 3B450C                      cmp          eax,[ebp][00C]
.00401009: 7E0D                        jle         .000401018 --↓1
.0040100B: 6800B04000                  push         00040B000 --↓2
.00401010: E8AA000000                  call        .0004010BF --↓3
.00401015: 83C404                      add          esp,4
.00401018: 8B4D08                     1mov          ecx,[ebp][8]
.0040101B: 3B4D0C                      cmp          ecx,[ebp][00C]
.0040101E: 750D                        jnz         .00040102D --↓4
.00401020: 6808B04000                  push         00040B008 ;'a==b' --↓5
.00401025: E895000000                  call        .0004010BF --↓3
.0040102A: 83C404                      add          esp,4
.0040102D: 8B5508                     4mov          edx,[ebp][8]
.00401030: 3B550C                      cmp          edx,[ebp][00C]
.00401033: 7D0D                        jge         .000401042 --↓6
.00401035: 6810B04000                  push         00040B010 --↓7
.0040103A: E880000000                  call        .0004010BF --↓3
.0040103F: 83C404                      add          esp,4
.00401042: 5D                         6pop          ebp
.00401043: C3                          retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401044: CC                          int          3
.00401045: CC                          int          3
.00401046: CC                          int          3
.00401047: CC                          int          3
.00401048: CC                          int          3
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam
```

Рис. 11.1: Hiew: функция `f_unsigned()`

Собственно, задач три:

- заставить первый переход срабатывать всегда;
- заставить второй переход не срабатывать никогда;
- заставить третий переход срабатывать всегда.

Так мы направим путь исполнения кода (code flow) во второй `printf()`, и он всегда будет срабатывать и выводить на консоль «a==b».

Для этого нужно изменить три инструкции (или байта):

- Первый переход теперь будет JMP, но смещение перехода (jump offset) останется прежним.
- Второй переход может быть и будет срабатывать иногда, но в любом случае он будет совершать переход только на следующую инструкцию, потому что мы выставляем смещение перехода (jump offset) в 0. В этих инструкциях смещение перехода просто прибавляется к адресу следующей инструкции. Когда смещение 0, переход будет на следующую инструкцию.
- Третий переход конвертируем в JMP точно так же, как и первый, он будет срабатывать всегда.

Что и делаем:

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe                 FWO EDITMODE          a32 PE   00000434 Hiew 8.02 (c)SEN
 00000400: 55                           push        ebp
 00000401: 8BEC                         mov         ebp,esp
 00000403: 8B4508                       mov         eax,[ebp][8]
 00000406: 3B450C                       cmp         eax,[ebp][00C]
 00000409: EB0D                         jmps        000000418
 0000040B: 6800B04000                   push        00040B000 ;' @  '
 00000410: E8AA000000                   call        0000004BF
 00000415: 83C404                       add         esp,4
 00000418: 8B4D08                       mov         ecx,[ebp][8]
 0000041B: 3B4D0C                       cmp         ecx,[ebp][00C]
 0000041E: 7500                         jnz         000000420
 00000420: 6808B04000                   push        00040B008 ;' @ '
 00000425: E895000000                   call        0000004BF
 0000042A: 83C404                       add         esp,4
 0000042D: 8B5508                       mov         edx,[ebp][8]
 00000430: 3B550C                       cmp         edx,[ebp][00C]
 00000433: EB0D                         jmps        000000442
 00000435: 6810B04000                   push        00040B010 ;' @ '
 0000043A: E880000000                   call        0000004BF
 0000043F: 83C404                       add         esp,4
 00000442: 5D                           pop         ebp
 00000443: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000444: CC                           int         3
 00000445: CC                           int         3
 00000446: CC                           int         3
 00000447: CC                           int         3
 00000448: CC                           int         3
 1     2NOPs  3     4     5     6     7     8Table 9     10    11    12
```

Рис. 11.2: Hiew: модифицируем функцию `f_unsigned()`

Если забыть про какой-то из переходов, то тогда будет срабатывать несколько вызовов `printf()`, а нам ведь нужно чтобы исполнялся только один.

11.2. Вычисление абсолютной величины

Это простая функция:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

11.2.1. Оптимизирующий MSVC

Обычный способ генерации кода:

Листинг 11.4: Оптимизирующий MSVC 2012 x64

```
i$ = 8
my_abs  PROC
; ECX = input
        test    ecx, ecx
; проверить знак входного значения
```

```
; пропустить инструкцию NEG если знак положительный
        jns     SHORT $LN2@my_abs
; поменять знак
        neg     ecx
$LN2@my_abs:
; подготовить результат в EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

11.3. Тернарный условный оператор

Тернарный условный оператор (ternary conditional operator) в Си/Си++ это:

```
expression ? expression : expression
```

И вот пример:

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

11.3.1. x86

Старые и неоптимизирующие компиляторы генерируют код так, как если бы выражение `if/else` было использовано вместо него:

Листинг 11.5: Неоптимизирующий MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; будет использовано как временная переменная
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; сравнить входное значение с 10
        cmp     DWORD PTR _a$[ebp], 10
; переход на $LN3@f если не равно
        jne     SHORT $LN3@f
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; перейти на выход
        jmp     SHORT $LN4@f
$LN3@f:
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; это выход. скопировать указатель на строку из временной переменной в EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Листинг 11.6: Оптимизирующий MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; сравнить входное значение с 10
```

```
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; переход на $LN4@f если равно
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

Новые компиляторы могут быть более краткими:

Листинг 11.7: Оптимизирующий MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; загрузить указатели на обе строки
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; сравнить входное значение с 10
        cmp     ecx, 10
; если равно, скопировать значение из RDX ("it is ten")
; если нет, ничего не делаем. указатель на строку "it is not ten" всё еще в RAX.
        cmove   rax, rdx
        ret     0
f       ENDP
```

Оптимизирующий GCC 4.8 для x86 также использует инструкцию CMOVcc, тогда как неоптимизирующий GCC 4.8 использует условные переходы.

11.3.2. Перепишем, используя обычный if/else

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

Интересно, оптимизирующий GCC 4.8 для x86 также может генерировать CMOVcc в этом случае:

Листинг 11.8: Оптимизирующий GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
.LFB0:
; сравнить входное значение с 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; если результат сравнение Not Equal (не равно), скопировать значение из EDX в EAX
; а если нет, то ничего не делать
        cmovne  eax, edx
        ret
```

Но оптимизирующий MSVC 2012 пока не так хорош.

11.4. Поиск минимального и максимального значения

11.4.1. 32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Листинг 11.9: Неоптимизирующий MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
        cmp     eax, DWORD PTR _b$[ebp]
; переход, если A больше или равно B:
        jge     SHORT $LN2@my_min
; перезагрузить A в EAX в противном случае и перейти на выход
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; это избыточная JMP
$LN2@my_min:
; возврат B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
        cmp     eax, DWORD PTR _b$[ebp]
; переход, если A меньше или равно B:
        jle     SHORT $LN2@my_max
; перезагрузить A в EAX в противном случае и перейти на выход
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; это избыточная JMP
$LN2@my_max:
; возврат B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

Эти две функции отличаются друг от друга только инструкцией условного перехода: JGE («Jump if Greater or Equal» — переход если больше или равно) используется в первой и JLE («Jump if Less or Equal» — переход если меньше или равно) во второй.

Здесь есть ненужная инструкция JMP в каждой функции, которую MSVC, наверное, оставил по ошибке.

11.5. Вывод

11.5.1. x86

Примерный скелет условных переходов:

Листинг 11.10: x86

```
CMP register, register/value
Jcc true ; cc=код условия
false:
... код, исполняющийся, если сравнение ложно ...
JMP exit
true:
... код, исполняющийся, если сравнение истинно ...
exit:
```

11.5.2. Без инструкций перехода

Если тело условного выражения очень короткое, может быть использована инструкция условного копирования: MOVcc в ARM (в режиме ARM), CSEL в ARM64, CMOVcc в x86.

Глава 12

switch()/case/default

12.1. Если вариантов мало

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

12.1.1. x86

Неоптимизирующий MSVC

Это дает в итоге (MSVC 2010):

Листинг 12.1: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
```

```
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f     ENDP
```

Наша функция со оператором switch(), с небольшим количеством вариантов, это практически аналог подобной конструкции:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Когда вариантов немного и мы видим подобный код, невозможно сказать с уверенностью, был ли в оригинальном исходном коде switch(), либо просто набор операторов if(). То есть, switch() это синтаксический сахар для большого количества вложенных проверок при помощи if().

В самом выходном коде ничего особо нового, за исключением того, что компилятор зачем-то перекладывает входящую переменную (a) во временную в локальном стеке v64[1].

Если скомпилировать это при помощи GCC 4.4.1, то будет почти то же самое, даже с максимальной оптимизацией (ключ -O3).

Оптимизирующий MSVC

Попробуем включить оптимизацию кодегенератора MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

Листинг 12.2: MSVC

```
_a$ = 8 ; size = 4
_f     PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
```

[1] Локальные переменные в стеке с префиксом tv — так MSVC называет внутренние переменные для своих нужд

```
    jmp     _printf
_f      ENDP
```

Вот здесь уже всё немного по-другому, причем не без грязных трюков.

Первое: a помещается в EAX и от него отнимается 0. Звучит абсурдно, но нужно это для того, чтобы проверить, 0 ли в EAX был до этого? Если да, то выставится флаг ZF (что означает, что результат вычитания 0 от числа стал 0) и первый условный переход JE (*Jump if Equal* или его синоним JZ — *Jump if Zero*) сработает на метку $LN4@f, где выводится сообщение 'zero'. Если первый переход не сработал, от значения отнимается по единице, и если на какой-то стадии в результате образуется 0, то сработает соответствующий переход.

И в конце концов, если ни один из условных переходов не сработал, управление передается printf() со строковым аргументом 'something unknown'.

Второе: мы видим две, мягко говоря, необычные вещи: указатель на сообщение помещается в переменную a, и затем printf() вызывается не через CALL, а через JMP. Объяснение этому простое. Вызывающая функция заталкивает в стек некоторое значение и через CALL вызывает нашу функцию. CALL в свою очередь заталкивает в стек адрес возврата (RA[2]) и делает безусловный переход на адрес нашей функции. Наша функция в самом начале (да и в любом её месте, потому что в теле функции нет ни одной инструкции, которая меняет что-то в стеке или в ESP) имеет следующую разметку стека:

- ESP — хранится RA
- ESP+4 — хранится значение a

С другой стороны, чтобы вызвать printf(), нам нужна почти такая же разметка стека, только в первом аргументе нужен указатель на строку. Что, собственно, этот код и делает.

Он заменяет свой первый аргумент на адрес строки, и затем передает управление printf(), как если бы вызвали не нашу функцию f(), а сразу printf(). printf() выводит некую строку на stdout, затем исполняет инструкцию RET, которая из стека достает RA и управление передается в ту функцию, которая вызывала f(), минуя при этом конец функции f().

Всё это возможно, потому что printf() вызывается в f() в самом конце. Всё это чем-то даже похоже на longjmp()[3]. И всё это, разумеется, сделано для экономии времени исполнения.

12.1.2. Вывод

Оператор *switch()* с малым количеством вариантов трудноотличим от применения конструкции *if/else*: листинг.12.1.1.

12.2. И если много

Если ветвлений слишком много, то генерировать слишком длинный код с многочисленными JE/JNE уже не так удобно.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

[2] Адрес возврата
[3] wikipedia

12.2.1. x86

Неоптимизирующий MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010):

Листинг 12.3: MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad    2 ; выровнять следующую метку
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

Здесь происходит следующее: в теле функции есть набор вызовов `printf()` с разными аргументами. Все они имеют, конечно же, адреса, а также внутренние символические метки, которые присвоил им компилятор. Также все эти метки указываются во внутренней таблице `$LN11@f`.

В начале функции, если a больше 4, то сразу происходит переход на метку `$LN1@f`, где вызывается `printf()` с аргументом `'something unknown'`.

А если a меньше или равно 4, то это значение умножается на 4 и прибавляется адрес таблицы с переходами (`$LN11@f`). Таким образом, получается адрес внутри таблицы, где лежит нужный адрес внутри тела функции. Например, возьмем

a равным 2. $2 * 4 = 8$ (ведь все элементы таблицы — это адреса внутри 32-битного процесса, таким образом, каждый элемент занимает 4 байта). 8 прибавить к `$LN11@f` — это будет элемент таблицы, где лежит `$LN4@f`. JMP вытаскивает из таблицы адрес `$LN4@f` и делает безусловный переход туда.

Эта таблица иногда называется *jumptable* или *branch table*[4].

А там вызывается `printf()` с аргументом `'two'`. Дословно, инструкция `jmp DWORD PTR $LN11@f[ecx*4]` означает *перейти по DWORD, который лежит по адресу* `$LN11@f + ecx * 4`.

это макрос ассемблера, выравнивающий начало таблицы, чтобы она располагалась по адресу кратному 4 (или 16). Это нужно для того, чтобы процессор мог эффективнее загружать 32-битные значения из памяти через шину с памятью, кэш-память, и т.д.

Неоптимизирующий GCC

Посмотрим, что сгенерирует GCC 4.4.1:

Листинг 12.4: GCC 4.4.1

```
       public f
f      proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       sub     esp, 18h
       cmp     [ebp+arg_0], 4
       ja      short loc_8048444
       mov     eax, [ebp+arg_0]
       shl     eax, 2
       mov     eax, ds:off_804855C[eax]
       jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
       mov     [esp+18h+var_18], offset aZero ; "zero"
       call    _puts
       jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
       mov     [esp+18h+var_18], offset aOne ; "one"
       call    _puts
       jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
       mov     [esp+18h+var_18], offset aTwo ; "two"
       call    _puts
       jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
       mov     [esp+18h+var_18], offset aThree ; "three"
       call    _puts
       jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
       mov     [esp+18h+var_18], offset aFour ; "four"
       call    _puts
       jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
       mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
       call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
       leave
       retn
f      endp
```

[4]Сам метод раньше назывался *computed GOTO* В ранних версиях FORTRAN: wikipedia. Не очень-то и полезно в наше время, но каков термин!

```
off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

Практически то же самое, за исключением мелкого нюанса: аргумент из `arg_0` умножается на 4 при помощи сдвига влево на 2 бита (это почти то же самое что и умножение на 4) (15.2.1 (стр. 61)). Затем адрес метки внутри функции берется из массива `off_804855C` и адресуется при помощи вычисленного индекса.

12.2.2. Вывод

Примерный скелет оператора *switch()*:

Листинг 12.5: x86

```
MOV REG, input
CMP REG, 4 ; максимальное количество меток
JA default
SHL REG, 2 ; найти элемент в таблице. сдвинуть на 3 бита в x64
MOV REG, jump_table[REG]
JMP REG

case1:
    ; делать что-то
    JMP exit
case2:
    ; делать что-то
    JMP exit
case3:
    ; делать что-то
    JMP exit
case4:
    ; делать что-то
    JMP exit
case5:
    ; делать что-то
    JMP exit

default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Переход по адресу из таблицы переходов может быть также реализован такой инструкцией: `JMP jump_table[REG*4]`. Или `JMP jump_table[REG*8]` в x64.

Таблица переходов (*jumptable*) это просто массив указателей, как это будет вскоре описано: 16.4 (стр. 68).

12.3. Когда много *case* в одном блоке

Вот очень часто используемая конструкция: несколько *case* может быть использовано в одном блоке:

```
#include <stdio.h>

void f(int a)
{
```

ГЛАВА 12. SWITCH()/CASE/DEFAULT

```
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
        default:
                printf ("default\n");
                break;
        };
};

int main()
{
        f(4);
};
```

Слишком расточительно генерировать каждый блок для каждого случая, поэтому обычно генерируется каждый блок плюс некий диспетчер.

12.3.1. MSVC

Листинг 12.6: Оптимизирующий MSVC 2010

```
$SG2798 DB      '1, 2, 7, 10', 0aH, 00H
$SG2800 DB      '3, 4, 5', 0aH, 00H
$SG2802 DB      '8, 9, 21', 0aH, 00H
$SG2804 DB      '22', 0aH, 00H
$SG2806 DB      'default', 0aH, 00H

_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        dec     eax
        cmp     eax, 21
        ja      SHORT $LN1@f
        movzx   eax, BYTE PTR $LN10@f[eax]
        jmp     DWORD PTR $LN11@f[eax*4]
$LN5@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
        jmp     DWORD PTR __imp__printf
$LN4@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
        jmp     DWORD PTR __imp__printf
$LN3@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
        jmp     DWORD PTR __imp__printf
$LN2@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
        jmp     DWORD PTR __imp__printf
$LN1@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
```

 ГЛАВА 12. SWITCH()/CASE/DEFAULT

```
29          jmp     DWORD PTR __imp__printf
30          npad    2 ; выровнять таблицу $LN11@f по 16-байтной границе
31  $LN11@f:
32          DD      $LN5@f ; вывести '1, 2, 7, 10'
33          DD      $LN4@f ; вывести '3, 4, 5'
34          DD      $LN3@f ; вывести '8, 9, 21'
35          DD      $LN2@f ; вывести '22'
36          DD      $LN1@f ; вывести 'default'
37  $LN10@f:
38          DB      0 ; a=1
39          DB      0 ; a=2
40          DB      1 ; a=3
41          DB      1 ; a=4
42          DB      1 ; a=5
43          DB      1 ; a=6
44          DB      0 ; a=7
45          DB      2 ; a=8
46          DB      2 ; a=9
47          DB      0 ; a=10
48          DB      4 ; a=11
49          DB      4 ; a=12
50          DB      4 ; a=13
51          DB      4 ; a=14
52          DB      4 ; a=15
53          DB      4 ; a=16
54          DB      4 ; a=17
55          DB      4 ; a=18
56          DB      4 ; a=19
57          DB      2 ; a=20
58          DB      2 ; a=21
59          DB      3 ; a=22
60  _f      ENDP
```

Здесь видим две таблицы: первая таблица (`$LN10@f`) это таблица индексов, а вторая таблица (`$LN11@f`) это массив указателей на блоки.

В начале, входное значение используется как индекс в таблице индексов (строка 13).

Вот краткое описание значений в таблице: 0 это первый блок *case* (для значений 1, 2, 7, 10), 1 это второй (для значений 3, 4, 5), 2 это третий (для значений 8, 9, 21), 3 это четвертый (для значений 22), 4 это для блока по умолчанию.

Мы получаем индекс для второй таблицы указателей на блоки и переходим туда (строка 14).

Ещё нужно отметить то, что здесь нет случая для нулевого входного значения. Поэтому мы видим инструкцию DEC на строке 10 и таблица начинается с $a = 1$. Потому что незачем выделять в таблице элемент для $a = 0$.

Это очень часто используемый шаблон.

В чем же экономия? Почему нельзя сделать так, как уже обсуждалось (12.2.1 (стр. 47)), используя только одну таблицу, содержащую указатели на блоки? Причина в том, что элементы в таблице индексов занимают только по 8-битному байту, поэтому всё это более компактно.

12.4. Fall-through

Ещё одно популярное использование оператора `switch()` это т.н. «fallthrough» («провал»). Вот простой пример:

```
#define R 1
#define W 2
#define RW 3

void f(int type)
{
        int read=0, write=0;

        switch (type)
        {
        case RW:
                read=1;
        case W:
                write=1;
```

```
15                  break;
16          case R:
17                  read=1;
18                  break;
19          default:
20                  break;
21          };
22          printf ("read=%d, write=%d\n", read, write);
23  };
```

Если $type = 1$ (R), $read$ будет выставлен в 1, если $type = 2$ (W), $write$ будет выставлен в 2. В случае $type = 3$ (RW), обе $read$ и $write$ будут выставлены в 1.

Фрагмент кода на строке 14 будет исполнен в двух случаях: если $type = RW$ или если $type = W$. Там нет «break» для «case RW», и это нормально.

12.4.1. MSVC x86

Листинг 12.7: MSVC 2012

```
$SG1305 DB      'read=%d, write=%d', 0aH, 00H

_write$ = -12   ; size = 4
_read$ = -8     ; size = 4
tv64 = -4       ; size = 4
_type$ = 8      ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _read$[ebp], 0
        mov     DWORD PTR _write$[ebp], 0
        mov     eax, DWORD PTR _type$[ebp]
        mov     DWORD PTR tv64[ebp], eax
        cmp     DWORD PTR tv64[ebp], 1 ; R
        je      SHORT $LN2@f
        cmp     DWORD PTR tv64[ebp], 2 ; W
        je      SHORT $LN3@f
        cmp     DWORD PTR tv64[ebp], 3 ; RW
        je      SHORT $LN4@f
        jmp     SHORT $LN5@f
$LN4@f: ; case RW:
        mov     DWORD PTR _read$[ebp], 1
$LN3@f: ; case W:
        mov     DWORD PTR _write$[ebp], 1
        jmp     SHORT $LN5@f
$LN2@f: ; case R:
        mov     DWORD PTR _read$[ebp], 1
$LN5@f: ; default
        mov     ecx, DWORD PTR _write$[ebp]
        push    ecx
        mov     edx, DWORD PTR _read$[ebp]
        push    edx
        push    OFFSET $SG1305 ; 'read=%d, write=%d'
        call    _printf
        add     esp, 12
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Код почти полностью повторяет то, что в исходнике. Там нет переходов между метками `$LN4@f` и `$LN3@f`: так что когда управление (code flow) находится на `$LN4@f`, $read$ в начале выставляется в 1, затем $write$. Наверное, поэтому всё это и называется «проваливаться»: управление проваливается через один фрагмент кода (выставляющий $read$) в другой (выставляющий $write$). Если $type = W$, мы оказываемся на `$LN3@f`, так что код выставляющий $read$ в 1 не исполнится.

Глава 13

Циклы

13.1. Простой пример

13.1.1. x86

Для организации циклов в архитектуре x86 есть старая инструкция LOOP. Она проверяет значение регистра ECX и если оно не 0, делает декремент ECX и переход по метке, указанной в операнде. Возможно, эта инструкция не слишком удобная, потому что уже почти не бывает современных компиляторов, которые использовали бы её. Так что если вы видите где-то LOOP, то с большой вероятностью это вручную написанный код на ассемблере.

Обычно, циклы на Си/Си++ создаются при помощи `for()`, `while()`, `do/while()`.

Начнем с `for()`.

Это выражение описывает инициализацию, условие, операцию после каждой итерации (инкремент/декремент) и тело цикла.

```
for (инициализация; условие; после каждой итерации)
{
    тело_цикла;
}
```

Примерно так же, генерируемый код и будет состоять из этих четырех частей.

Возьмем пример:

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                printing_function(i);

        return 0;
};
```

Имеем в итоге (MSVC 2010):

Листинг 13.1: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; инициализация цикла
    jmp    SHORT $LN3@main
```

```
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; то что мы делаем после каждой итерации:
    add    eax, 1                  ; добавляем 1 к i
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; это условие проверяется *перед* каждой итерацией
    jge    SHORT $LN1@main         ; если i больше или равно 10, заканчиваем цикл
    mov    ecx, DWORD PTR _i$[ebp] ; тело цикла: вызов функции printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; переход на начало цикла
$LN1@main:                         ; конец цикла
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

В принципе, ничего необычного.

Листинг 13.2: Оптимизирующий MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10      ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Здесь происходит следующее: переменную i компилятор не выделяет в локальном стеке, а выделяет целый регистр под нее: ESI. Это возможно для маленьких функций, где мало локальных переменных.

В принципе, всё то же самое, только теперь одна важная особенность: `f()` не должна менять значение ESI. Наш компилятор уверен в этом, а если бы и была необходимость использовать регистр ESI в функции `f()`, то её значение сохранялось бы в стеке. Примерно так же как и в нашем листинге: обратите внимание на `PUSH ESI/POP ESI` в начале и конце функции.

13.1.2. Ещё кое-что

По генерируемому коду мы видим следующее: после инициализации i, тело цикла не исполняется. Исполняется сразу проверка условия i, а лишь затем исполняется тело цикла. Это правильно. Потому что если условие в самом начале не выполняется, тело цикла исполнять нельзя. Так может быть, например, в таком случае:

```
for (i=0; i<total_entries_to_process; i++)
    тело_цикла;
```

Если *total_entries_to_process* равно 0, тело цикла не должно исполниться ни разу. Поэтому проверка условия происходит перед тем как исполнить само тело.

Впрочем, оптимизирующий компилятор может переставить проверку условия и тело цикла местами, если он уверен, что описанная здесь ситуация невозможна, как в случае с нашим простейшим примером и компиляторами Keil, Xcode (LLVM), MSVC и GCC в режиме оптимизации.

13.2. Функция копирования блоков памяти

Настоящие функции копирования памяти могут копировать по 4 или 8 байт на каждой итерации, использовать SIMD[1], векторизацию, и т.д. Но ради простоты, этот пример настолько прост, насколько это возможно.

[1] Single instruction, multiple data

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

13.2.1. Простейшая реализация

Листинг 13.3: GCC 4.9 x64 оптимизация по размеру (-Os)

```
my_memcpy:
; RDI = целевой адрес
; RSI = исходный адрес
; RDX = размер блока

; инициализировать счетчик (i) в 0
        xor     eax, eax
.L2:
; все байты скопированы? тогда заканчиваем:
        cmp     rax, rdx
        je      .L5
; загружаем байт по адресу RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; записываем байт по адресу RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

13.3. Вывод

Примерный скелет цикла от 2 до 9 включительно:

Листинг 13.4: x86

```
    mov [counter], 2 ; инициализация
    jmp check
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в локальном стеке
    add [counter], 1 ; инкремент
check:
    cmp [counter], 9
    jle body
```

Операция инкремента может быть представлена как 3 инструкции в неоптимизированном коде:

Листинг 13.5: x86

```
    MOV [counter], 2 ; инициализация
    JMP check
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в локальном стеке
    MOV REG, [counter] ; инкремент
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

Если тело цикла короткое, под переменную счетчика можно выделить целый регистр:

Листинг 13.6: x86

```
    MOV EBX, 2 ; инициализация
    JMP check
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в EBX, но не изменяем её!
    INC EBX ; инкремент
check:
    CMP EBX, 9
    JLE body
```

Некоторые части цикла могут быть сгенерированы компилятором в другом порядке:

Листинг 13.7: x86

```
    MOV [counter], 2 ; инициализация
    JMP label_check
label_increment:
    ADD [counter], 1 ; инкремент
label_check:
    CMP [counter], 10
    JGE exit
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в локальном стеке
    JMP label_increment
exit:
```

Обычно условие проверяется *перед* телом цикла, но компилятор может перестроить цикл так, что условие проверяется *после* тела цикла. Это происходит тогда, когда компилятор уверен, что условие всегда будет *истинно* на первой итерации, так что тело цикла исполнится как минимум один раз:

Листинг 13.8: x86

```
    MOV REG, 2 ; инициализация
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в REG, но не изменяем её!
    INC REG ; инкремент
    CMP REG, 10
    JL body
```

Используя инструкцию LOOP. Это редкость, компиляторы не используют её. Так что если вы её видите, это верный знак, что этот фрагмент кода написан вручную:

Листинг 13.9: x86

```
    ; считать от 10 до 1
    MOV ECX, 10
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в ECX, но не изменяем её!
    LOOP body
```

Глава 14

Простая работа с Си-строками

14.1. strlen()

Ещё немного о циклах. Часто функция `strlen()`[1] реализуется при помощи `while()`. Например, вот как это сделано в стандартных библиотеках MSVC:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

14.1.1. x86

Неоптимизирующий MSVC

Итак, компилируем:

```
_eos$ = -4                          ; size = 4
_str$ = 8                           ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; взять указатель на символ из "str"
    mov     DWORD PTR _eos$[ebp], eax  ; и переложить его в нашу локальную переменную "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; взять байт, на который указывает ECX и положить его в EDX расширяя до 32-х бит, учитывая знак

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; инкремент EAX
    mov     DWORD PTR _eos$[ebp], eax  ; положить eax назад в "eos"
    test    edx, edx                   ; EDX ноль?
    je      SHORT $LN1@strlen_         ; да, то что лежит в EDX это ноль, выйти из цикла
    jmp     SHORT $LN2@strlen_         ; продолжаем цикл
$LN1@strlen_:
```

[1] подсчет длины строки в Си

```
    ; здесь мы вычисляем разницу двух указателей

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                      ; отнимаем от разницы еще единицу и возвращаем результат
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Здесь две новых инструкции: MOVSX и TEST.

О первой. MOVSX предназначена для того, чтобы взять байт из какого-либо места в памяти и положить его, в нашем случае, в регистр EDX. Но регистр EDX — 32-битный. MOVSX означает *MOV with Sign-Extend*. Оставшиеся биты с 8-го по 31-й MOVSX сделает единицей, если исходный байт в памяти имеет знак *минус*, или заполнит нулями, если знак *плюс*.

И вот зачем всё это.

По умолчанию в MSVC и GCC тип *char* — знаковый. Если у нас есть две переменные, одна *char*, а другая *int* (*int* тоже знаковый), и если в первой переменной лежит -2 (что кодируется как 0xFE) и мы просто переложим это в *int*, то там будет 0x000000FE, а это, с точки зрения *int*, даже знакового, будет 254, но никак не -2. -2 в переменной *int* кодируется как 0xFFFFFFFE. Для того чтобы значение 0xFE из переменной типа *char* переложить в знаковый *int* с сохранением всего, нужно узнать его знак и затем заполнить остальные биты. Это делает MOVSX.

См. также об этом раздел «*Представление знака в числах*» (22 (стр. 105)).

Хотя конкретно здесь компилятору вряд ли была особая надобность хранить значение *char* в регистре EDX, а не его восьмибитной части, скажем DL. Но получилось, как получилось. Должно быть register allocator компилятора сработал именно так.

Позже выполняется TEST EDX, EDX. Об инструкции TEST читайте в разделе о битовых полях (17 (стр. 75)). Конкретно здесь эта инструкция просто проверяет состояние регистра EDX на 0.

Оптимизирующий MSVC

Теперь скомпилируем всё то же самое в MSVC 2012, но с включенной оптимизацией (/Ox) :

Листинг 14.1: Оптимизирующий MSVC 2012 /Ob0

```
_str$ = 8                       ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> указатель на строку
        mov     eax, edx                    ; переложить в EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; нет, продолжаем цикл
        sub     eax, edx                    ; вычисляем разницу указателей
        dec     eax                         ; декремент EAX
        ret     0
_strlen ENDP
```

Здесь всё попроще стало. Но следует отметить, что компилятор обычно может так хорошо использовать регистры только на небольших функциях с небольшим количеством локальных переменных.

INC/DEC — это инструкции инкремента-декремента. Попросту говоря — увеличить на единицу или уменьшить.

Глава 15

Замена одних арифметических инструкций на другие

В целях оптимизации одна инструкция может быть заменена другой, или даже группой инструкций. Например, ADD и SUB могут заменять друг друга: строка 18 в листинг.**??**.

15.1. Умножение

15.1.1. Умножение при помощи сложения

Вот простой пример:

Листинг 15.1: Оптимизирующий MSVC 2010

```
unsigned int f(unsigned int a)
{
        return a*8;
};
```

Умножение на 8 заменяется на три инструкции сложения, делающих то же самое. Должно быть, оптимизатор в MSVC решил, что этот код может быть быстрее.

```
_TEXT   SEGMENT
_a$ = 8                                                 ; size = 4
_f      PROC
; File c:\polygon\c\2.c
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

15.1.2. Умножение при помощи сдвигов

Ещё очень часто умножения и деления на числа вида 2^n заменяются на инструкции сдвигов.

```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

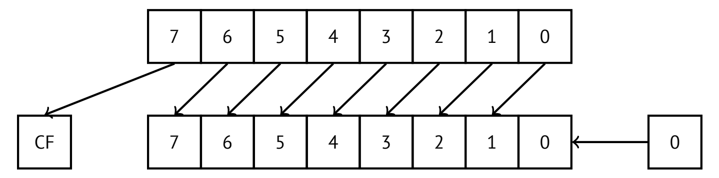
Листинг 15.2: Неоптимизирующий MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
```

```
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

Умножить на 4 это просто сдвинуть число на 2 бита влево, вставив 2 нулевых бита справа (как два самых младших бита). Это как умножить 3 на 100 — нужно просто дописать два нуля справа.

Вот как работает инструкция сдвига влево:



Добавленные биты справа — всегда нули.

15.1.3. Умножение при помощи сдвигов, сложений и вычитаний

Можно избавиться от операции умножения, если вы умножаете на числа вроде 7 или 17, и использовать сдвиги. Здесь используется относительно простая математика.

32-бита

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Листинг 15.3: Оптимизирующий MSVC 2012

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
```

```
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

64-бита

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

Листинг 15.4: Оптимизирующий MSVC 2012

```
; a*7
f1:
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
```

```
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

15.2. Деление

15.2.1. Деление используя сдвиги

Например, возьмем деление на 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

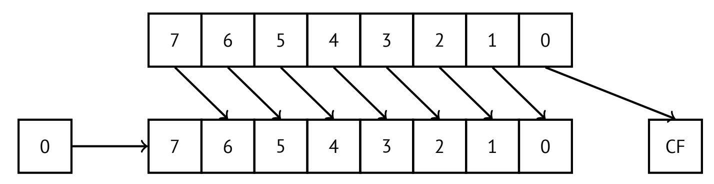
Имеем в итоге (MSVC 2010):

Листинг 15.5: MSVC 2010

```
_a$ = 8                                                 ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

Инструкция SHR (*SHift Right*) в данном примере сдвигает число на 2 бита вправо. При этом освободившиеся два бита слева (т.е. самые старшие разряды) выставляются в нули. А самые младшие 2 бита выкидываются. Фактически, эти два выкинутых бита — остаток от деления.

Инструкция SHR работает так же как и SHL, только в другую сторону.



Для того, чтобы это проще понять, представьте себе десятичную систему счисления и число 23. 23 можно разделить на 10 просто откинув последний разряд (3 — это остаток от деления). После этой операции останется 2 как частное.

Так что остаток выбрасывается, но это нормально, мы все-таки работаем с целочисленными значениями, а не с вещественными!

Глава 16

Массивы

Массив это просто набор переменных в памяти, обязательно лежащих рядом и обязательно одного типа[1].

16.1. Простой пример

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

16.1.1. x86

MSVC

Компилируем:

Листинг 16.1: MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                        ; size = 4
_a$ = -80                        ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84        ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN4@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
```

[1]AKA[2] «гомогенный контейнер»

```
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    mov    edx, DWORD PTR _a$[ebp+ecx*4]
    push   edx
    mov    eax, DWORD PTR _i$[ebp]
    push   eax
    push   OFFSET $SG2463
    call   _printf
    add    esp, 12         ; 0000000cH
    jmp    SHORT $LN2@main
$LN1@main:
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Ничего особенного, просто два цикла. Один изменяет массив, второй печатает его содержимое. Команда `shl ecx, 1` используется для умножения ECX на 2, об этом ниже (15.2.1 (стр. 61)).

Под массив выделено в стеке 80 байт, это 20 элементов по 4 байта.

16.2. Переполнение буфера

16.2.1. Чтение за пределами массива

Итак, индексация массива — это просто *массив[индекс]*. Если вы присмотритесь к коду, в цикле печати значений массива через `printf()` вы не увидите проверок индекса, *меньше ли он двадцати?* А что будет если он будет 20 или больше? Эта одна из особенностей Си/Си++, за которую их, собственно, и ругают.

Вот код, который и компилируется и работает:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[20]=%d\n", a[20]);

        return 0;
};
```

Вот результат компиляции в (MSVC 2008):

Листинг 16.2: Неоптимизирующий MSVC 2008

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main     PROC
    push   ebp
```

```
    mov    ebp, esp
    sub    esp, 84
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main:
    mov    eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Данный код при запуске выдал вот такой результат:

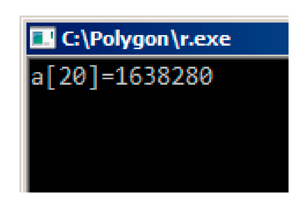


Рис. 16.1: OllyDbg: вывод в консоль

Это просто *что-то*, что волею случая лежало в стеке рядом с массивом, через 80 байт от его первого элемента.

16.2.2. Запись за пределы массива

Итак, мы прочитали какое-то число из стека явно *нелегально*, а что если мы запишем?

Вот что мы пишем:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

MSVC

И вот что имеем на ассемблере:

Листинг 16.3: Неоптимизирующий MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30 ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]      ; явный промах компилятора. эта инструкция лишняя.
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; а здесь в качестве второго операнда подошел бы ECX.
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Запускаете скомпилированную программу, и она падает. Немудрено. Но давайте теперь узнаем, где именно.

Загружаем в OllyDbg, трассируем пока запишутся все 30 элементов:

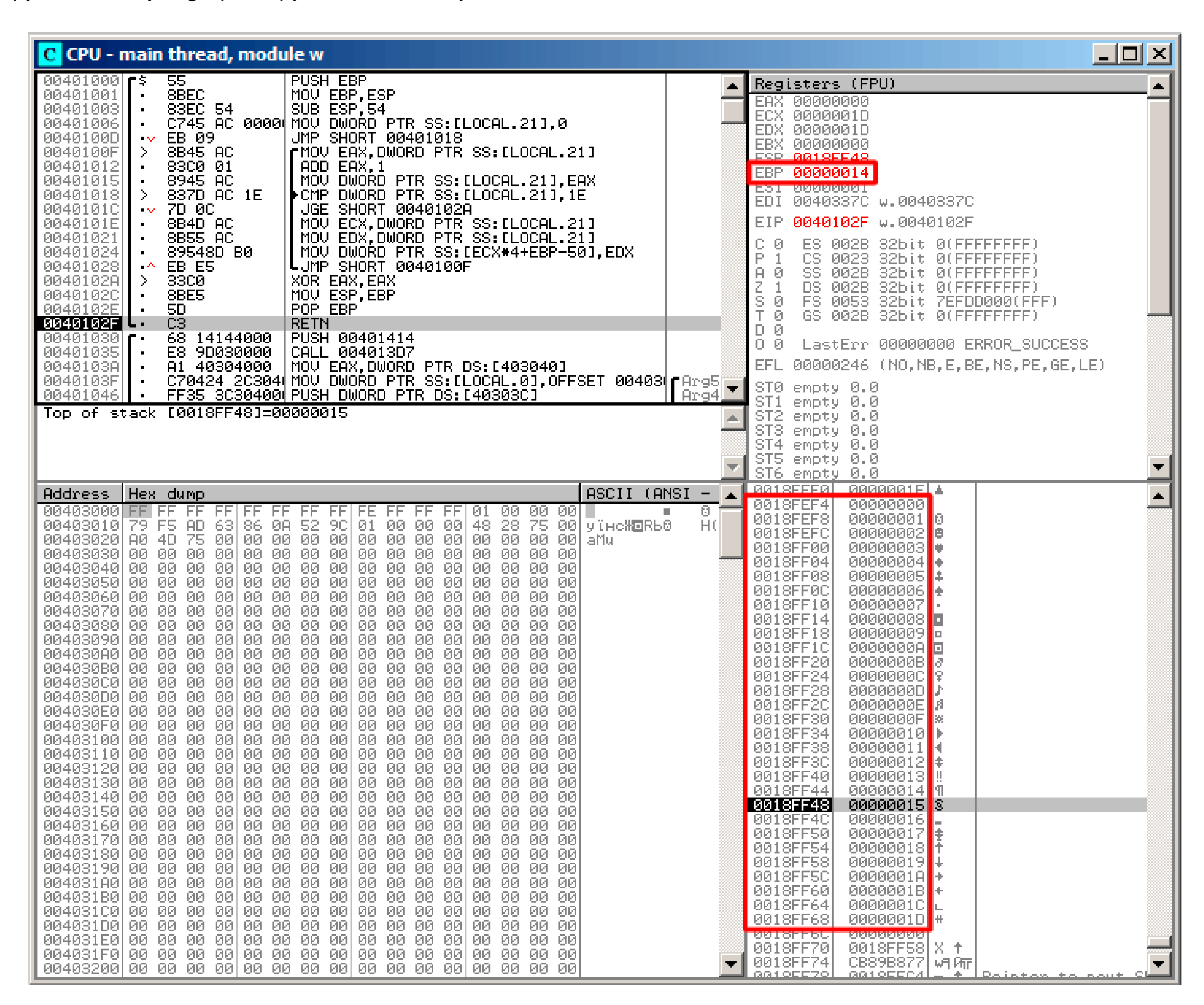


Рис. 16.2: OllyDbg: после восстановления EBP

Доходим до конца функции:

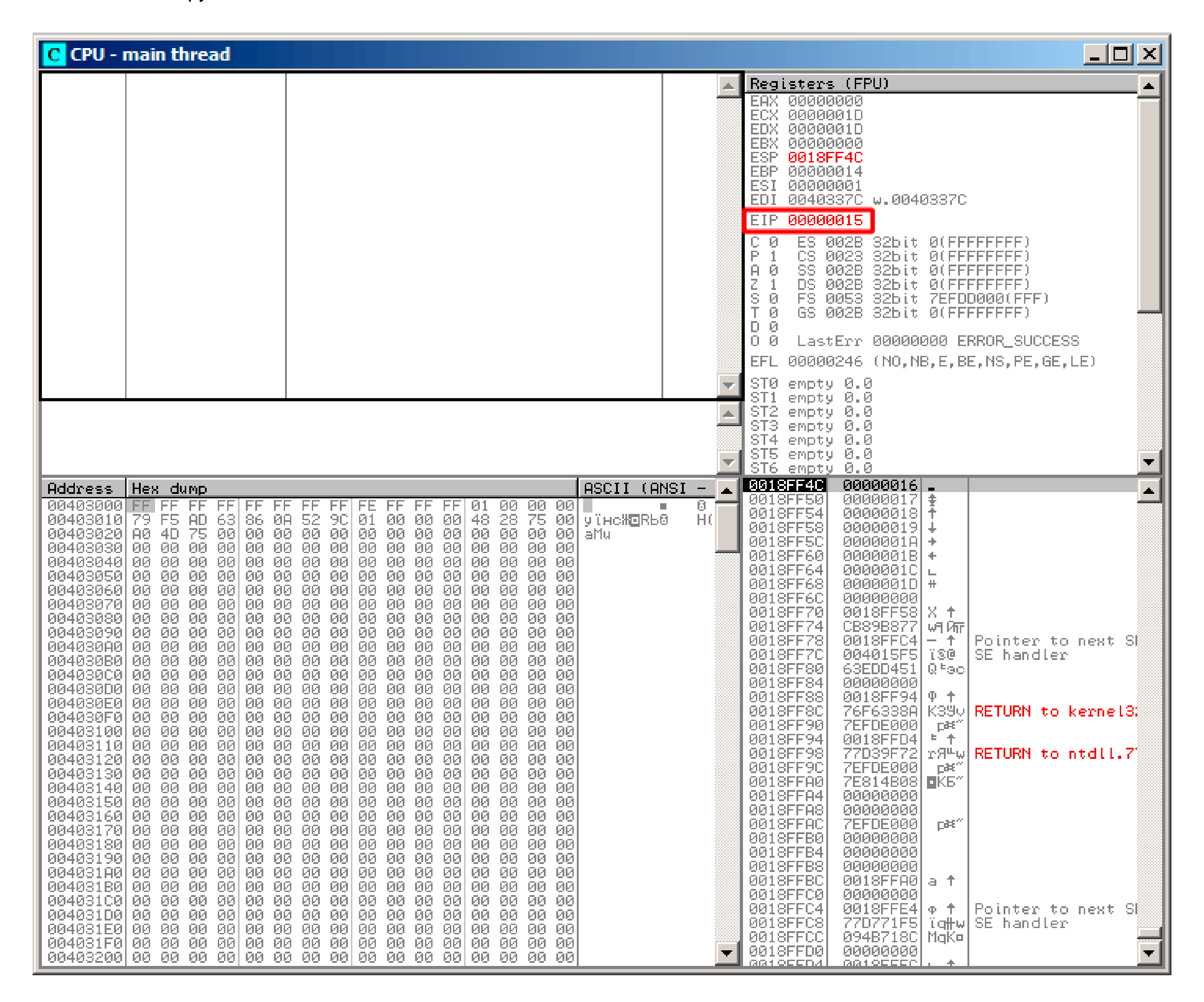


Рис. 16.3: OllyDbg: EIP восстановлен, но OllyDbg не может дизассемблировать по адресу 0x15

Итак, следите внимательно за регистрами.

EIP теперь 0x15. Это явно нелегальный адрес для кода — по крайней мере, win32-кода! Мы там как-то очутились, причем, сами того не хотели. Интересен также тот факт, что в EBP хранится 0x14, а в ECX и EDX — 0x1D.

Ещё немного изучим разметку стека.

После того как управление передалось в `main()`, в стек было сохранено значение EBP. Затем для массива и переменной i было выделено 84 байта. Это `(20+1)*sizeof(int)`. ESP сейчас указывает на переменную `_i` в локальном стеке и при исполнении следующего `PUSH что-либо`, *что-либо* появится рядом с `_i`.

Вот так выглядит разметка стека пока управление находится внутри `main()`:

ESP	4 байта выделенных для переменной i
ESP+4	80 байт выделенных для массива `a[20]`
ESP+84	сохраненное значение EBP
ESP+88	адрес возврата

Выражение `a[19]=что_нибудь` записывает последний *int* в пределах массива (пока что в пределах!)

Выражение `a[20]=что_нибудь` записывает *что_нибудь* на место где сохранено значение EBP.

Обратите внимание на состояние регистров на момент падения процесса. В нашем случае в 20-й элемент записалось значение 20. И вот всё дело в том, что заканчиваясь, эпилог функции восстанавливал значение EBP (20 в десятичной системе это как раз 0x14 в шестнадцатеричной). Далее выполнилась инструкция RET, которая на самом деле эквивалентна `POP EIP`.

Инструкция RET вытащила из стека адрес возврата (это адрес где-то внутри CRT), которая вызвала `main()`), а там было записано 21 в десятичной системе, то есть 0x15 в шестнадцатеричной. И вот процессор оказался по адресу 0x15, но исполняемого кода там нет, так что случилось исключение.

Добро пожаловать! Это называется *buffer overflow*[3].

Замените массив *int* на строку (массив *char*), нарочно создайте слишком длинную строку, передайте её в ту программу, в ту функцию, которая не проверяя длину строки скопирует её в слишком короткий буфер, и вы сможете указать программе, по какому именно адресу перейти. Не всё так просто в реальности, конечно, но началось всё с этого[4].

16.3. Еще немного о массивах

Теперь понятно, почему нельзя написать в исходном коде на Си/Си++ что-то вроде:

```
void f(int size)
{
    int a[size];
...
};
```

Чтобы выделить место под массив в локальном стеке, компилятору нужно знать размер массива, чего он на стадии компиляции, разумеется, знать не может.

Если вам нужен массив произвольной длины, то выделите столько, сколько нужно, через `malloc()`, а затем обращайтесь к выделенному блоку байт как к массиву того типа, который вам нужен.

Либо используйте возможность стандарта C99 [ISO07, с. 6.7.5/2], и внутри это очень похоже на alloca() (5.2.4 (стр. 11)).

Для работы в с памятью, можно также воспользоваться библиотекой сборщика мусора в Си. А для языка Си++ есть библиотеки с поддержкой умных указателей.

16.4. Массив указателей на строки

Вот пример массива указателей.

Листинг 16.4: Получить имя месяца

```
#include <stdio.h>

const char* month1[]=
{
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
};

// в пределах 0..11
const char* get_month1 (int month)
{
        return month1[month];
};
```

16.4.1. x64

Листинг 16.5: Оптимизирующий MSVC 2013 x64

```
_DATA   SEGMENT
month1  DQ      FLAT:$SG3122
```

[3] wikipedia
[4] Классическая статья об этом: [One96]

```
        DQ      FLAT:$SG3123
        DQ      FLAT:$SG3124
        DQ      FLAT:$SG3125
        DQ      FLAT:$SG3126
        DQ      FLAT:$SG3127
        DQ      FLAT:$SG3128
        DQ      FLAT:$SG3129
        DQ      FLAT:$SG3130
        DQ      FLAT:$SG3131
        DQ      FLAT:$SG3132
        DQ      FLAT:$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

Код очень простой:

- Первая инструкция MOVSXD копирует 32-битное значение из ECX (где передается аргумент :$month$) в RAX с знаковым расширением (потому что аргумент $month$ имеет тип *int*). Причина расширения в том, что это значение будет использоваться в вычислениях наряду с другими 64-битными значениями. Таким образом, оно должно быть расширено до 64-битного[5].
- Затем адрес таблицы указателей загружается в RCX.
- В конце концов, входное значение ($month$) умножается на 8 и прибавляется к адресу. Действительно: мы в 64-битной среде и все адреса (или указатели) требуют для хранения именно 64 бита (или 8 байт). Следовательно, каждый элемент таблицы имеет ширину в 8 байт. Вот почему для выбора элемента под нужным номером нужно пропустить $month * 8$ байт от начала. Это то, что делает MOV. Эта инструкция также загружает элемент по этому адресу. Для 1, элемент будет указателем на строку, содержащую «February», и т.д.

Оптимизирующий GCC 4.9 может это сделать даже лучше[6]:

Листинг 16.6: Оптимизирующий GCC 4.9 x64

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

32-bit MSVC

Скомпилируем также в 32-битном компиляторе MSVC:

Листинг 16.7: Оптимизирующий MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
```

[5] Это немного странная вещь, но отрицательный индекс массива может быть передан как $month$. И если так будет, отрицательное значение типа *int* будет расширено со знаком корректно и соответствующий элемент перед таблицей будет выбран. Всё это не будет корректно работать без знакового расширения.

[6] В листинге осталось «0+», потому что вывод ассемблера GCC не так скрупулёзен, чтобы убрать это. Это *displacement* и он здесь нулевой.

```
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

Входное значение не нужно расширять до 64-битного значения, так что оно используется как есть. И оно умножается на 4, потому что элементы таблицы имеют ширину 32 бита или 4 байта.

16.5. Многомерные массивы

Внутри многомерный массив выглядит так же как и линейный. Ведь память компьютера линейная, это одномерный массив. Но для удобства этот одномерный массив легко представить как многомерный.

К примеру, вот как элементы массива 3x4 расположены в одномерном массиве из 12 ячеек:

Смещение в памяти	элемент массива
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Таблица 16.1: Двухмерный массив представляется в памяти как одномерный

Вот по каким адресам в памяти располагается каждая ячейка двухмерного массива 3*4:

0	1	2	3
4	5	6	7
8	9	10	11

Таблица 16.2: Адреса в памяти каждой ячейки двухмерного массива

Чтобы вычислить адрес нужного элемента, сначала умножаем первый индекс (строку) на 4 (ширину массива), затем прибавляем второй индекс (столбец). Это называется *row-major order*, и такой способ представления массивов и матриц используется по крайней мере в Си/Си++ и Python. Термин *row-major order* означает по-русски примерно следующее: «сначала записываем элементы первой строки, затем второй, … и элементы последней строки в самом конце».

Другой способ представления называется *column-major order* (индексы массива используются в обратном порядке) и это используется по крайней мере в FORTRAN, MATLAB и R. Термин *column-major order* означает по-русски следующее: «сначала записываем элементы первого столбца, затем второго, … и элементы последнего столбца в самом конце».

Какой из способов лучше? В терминах производительности и кэш-памяти, лучший метод организации данных это тот, при котором к данным обращаются последовательно. Так что если ваша функция обращается к данным построчно, то *row-major order* лучше, и наоборот.

16.5.1. Пример с двумерным массивов

Мы будем работать с массивом типа *char*. Это значит, что каждый элемент требует только одного байта в памяти.

Пример с заполнением строки

Заполняем вторую строку значениями 0..3:

Листинг 16.8: Пример с заполнением строки

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // очистить массив
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // заполнить вторую строку значениями 0..3:
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

Все три строки обведены красным. Видно, что во второй теперь имеются байты 0, 1, 2 и 3:

Address	Hex dump
00C33370	00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380	02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 16.4: OllyDbg: массив заполнен

Пример с заполнением столбца

Заполняем третий столбец значениями 0..2:

Листинг 16.9: Пример с заполнением столбца

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // очистить массив
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // заполнить третий столбец значениями 0..2:
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

Здесь также обведены красным три строки. Видно, что в каждой строке, на третьей позиции, теперь записаны 0, 1 и 2.

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E AA EF 31 1E AA EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 16.5: OllyDbg: массив заполнен

16.5.2. Работа с двухмерным массивом как с одномерным

Мы можем легко убедиться, что можно работать с двухмерным массивом как с одномерным, используя по крайней мере два метода:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // обращаться с входным массивом как с одномерным
        // 4 здесь это ширина массива
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
        // обращаться с входным массивом как с указателем,
        // вычислить адрес, получить значение оттуда
        // 4 здесь это ширина массива
        return *(array+a*4+b);
};

int main()
{
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Компилируете и запускаете: мы увидим корректные значения.

Очарователен результат работы MSVC 2013 — все три процедуры одинаковые!

Листинг 16.10: Оптимизирующий MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=адрес массива
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+адрес массива
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=загрузить байт по адресу RAX+R9*4=b+адрес массива+a*4=адрес массива+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP
```

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

16.5.3. Пример с трехмерным массивом

То же самое и для многомерных массивов. На этот раз будем работать с массивом типа *int*: каждый элемент требует 4 байта в памяти.

Попробуем:

Листинг 16.11: простой пример

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

В итоге (MSVC 2010):

Листинг 16.12: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_z$ = 16                ; size = 4
_value$ = 20            ; size = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=значение
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

В принципе, ничего удивительного. В `insert()` для вычисления адреса нужного элемента массива три входных аргумента перемножаются по формуле $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, чтобы представить массив трехмерным. Не забывайте также, что тип *int* 32-битный (4 байта), поэтому все коэффициенты нужно умножить на 4.

Листинг 16.13: GCC 4.4.1

```
                public insert
```

```
insert          proc near

x               = dword ptr  8
y               = dword ptr  0Ch
z               = dword ptr  10h
value           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+x]
                mov     eax, [ebp+y]
                mov     ecx, [ebp+z]
                lea     edx, [eax+eax]              ; edx=y*2
                mov     eax, edx                    ; eax=y*2
                shl     eax, 4                      ; eax=(y*2)<<4 = y*2*16 = y*32
                sub     eax, edx                    ; eax=y*32 - y*2=y*30
                imul    edx, ebx, 600               ; edx=x*600
                add     eax, edx                    ; eax=eax+edx=y*30 + x*600
                lea     edx, [eax+ecx]              ; edx=y*30 + x*600 + z
                mov     eax, [ebp+value]
                mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=значение
                pop     ebx
                pop     ebp
                retn
insert          endp
```

Компилятор GCC решил всё сделать немного иначе. Для вычисления одной из операций ($30y$), GCC создал код, где нет самой операции умножения. Происходит это так: $(y+y) \ll 4-(y+y) = (2y) \ll 4-2y = 2\cdot16\cdot y-2y = 32y-2y = 30y$. Таким образом, для вычисления $30y$ используется только операция сложения, операция битового сдвига и операция вычитания. Это работает быстрее.

16.6. Вывод

Массив это просто набор значений в памяти, расположенных рядом друг с другом. Это справедливо для любых типов элементов, включая структуры. Доступ к определенному элементу массива это просто вычисление его адреса.

Глава 17

Работа с отдельными битами

Немало функций задают различные флаги в аргументах при помощи битовых полей[1]. Наверное, вместо этого можно было бы использовать набор переменных типа *bool*, но это было бы не очень экономно.

17.1. Проверка какого-либо бита

17.1.1. x86

Например в Win32 API:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS, ↙
    ↘ FILE_ATTRIBUTE_NORMAL, NULL);
```

Получаем (MSVC 2010):

Листинг 17.1: MSVC 2010

```
        push    0
        push    128                                     ; 00000080H
        push    4
        push    0
        push    1
        push    -1073741824                             ; c0000000H
        push    OFFSET $SG78813
        call    DWORD PTR __imp__CreateFileA@28
        mov     DWORD PTR _fh$[ebp], eax
```

Заглянем в файл WinNT.h:

Листинг 17.2: WinNT.h

```
#define GENERIC_READ                    (0x80000000L)
#define GENERIC_WRITE                   (0x40000000L)
#define GENERIC_EXECUTE                 (0x20000000L)
#define GENERIC_ALL                     (0x10000000L)
```

Всё ясно, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, и это значение используется как второй аргумент для функции `CreateFile()`[2].

Как `CreateFile()` будет проверять флаги? Заглянем в KERNEL32.DLL от Windows XP SP3 x86 и найдем в функции `CreateFileW()` в том числе и такой фрагмент кода:

Листинг 17.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429                  test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D                  mov     [ebp+var_8], 1
.text:7C83D434                  jz      short loc_7C83D417
.text:7C83D436                  jmp     loc_7C810817
```

[1] bit fields в англоязычной литературе
[2] msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

Здесь мы видим инструкцию TEST. Впрочем, она берет не весь второй аргумент функции, а только его самый старший байт (ebp+dwDesiredAccess+3) и проверяет его на флаг 0x40 (имеется ввиду флаг GENERIC_WRITE). TEST это то же что и AND, только без сохранения результата (вспомните что CMP это то же что и SUB, только без сохранения результатов (7.3.1 (стр. 23))).

Логика данного фрагмента кода примерно такая:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Если после операции AND останется этот бит, то флаг ZF не будет поднят и условный переход JZ не сработает. Переход возможен, только если в переменной dwDesiredAccess отсутствует бит 0x40000000 — тогда результат AND будет 0, флаг ZF будет поднят и переход сработает.

17.2. Установка и сброс отдельного бита

Например:

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

17.2.1. x86

Неоптимизирующий MSVC

Имеем в итоге (MSVC 2010):

Листинг 17.4: MSVC 2010

```
_rt$ = -4         ; size = 4
_a$ = 8           ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384                  ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                  ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f  ENDP
```

Инструкция OR здесь устанавливает в переменной ещё один бит, игнорируя остальные.

А AND сбрасывает некий бит. Можно также сказать, что AND здесь копирует все биты, кроме одного. Действительно, во втором операнде AND выставлены в единицу те биты, которые нужно сохранить, кроме одного, копировать который мы не хотим (и который 0 в битовой маске). Так проще понять и запомнить.

Оптимизирующий MSVC

Если скомпилировать в MSVC с оптимизацией (/Ox), то код еще короче:

Листинг 17.5: Оптимизирующий MSVC

```
_a$ = 8                      ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    and    eax, -513          ; fffffdffH
    or     eax, 16384         ; 00004000H
    ret    0
_f    ENDP
```

17.3. Сдвиги

Битовые сдвиги в Си/Си++ реализованы при помощи операторов $\ll$ и $\gg$.

В x86 есть инструкции SHL (SHift Left) и SHR (SHift Right) для этого.

Инструкции сдвига также активно применяются при делении или умножении на числа-степени двойки: 2^n (т.е. 1, 2, 4, 8, и т.д.): 15.1.2 (стр. 58), 15.2.1 (стр. 61).

Операции сдвига ещё потому так важны, потому что они часто используются для изолирования определенного бита или для конструирования значения из нескольких разрозненных бит.

17.4. Подсчет выставленных бит

Вот этот несложный пример иллюстрирует функцию, считающую количество бит-единиц во входном значении.

Эта операция также называется «population count»[3].

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

В этом цикле счетчик итераций i считает от 0 до 31, а $1 \ll i$ будет от 1 до 0x80000000. Описывая это словами, можно сказать *сдвинуть единицу на n бит влево*. Т.е. в некотором смысле, выражение $1 \ll i$ последовательно выдает все возможные позиции бит в 32-битном числе. Освободившийся бит справа всегда обнуляется.

Вот таблица всех возможных значений $1 \ll i$ для $i = 0 \ldots 31$:

[3] современные x86-процессоры (поддерживающие SSE4) даже имеют инструкцию POPCNT для этого

ГЛАВА 17. РАБОТА С ОТДЕЛЬНЫМИ БИТАМИ

Выражение в Си/Си++	Степень двойки	Десятичная форма	Шестнадцатеричная форма
$1 \ll 0$	1	1	1
$1 \ll 1$	2^1	2	2
$1 \ll 2$	2^2	4	4
$1 \ll 3$	2^3	8	8
$1 \ll 4$	2^4	16	0x10
$1 \ll 5$	2^5	32	0x20
$1 \ll 6$	2^6	64	0x40
$1 \ll 7$	2^7	128	0x80
$1 \ll 8$	2^8	256	0x100
$1 \ll 9$	2^9	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

Это числа-константы (битовые маски), которые крайне часто попадаются в практике reverse engineer-а, и их нужно уметь распознавать. Числа в десятичном виде заучивать, пожалуй, незачем, а числа в шестнадцатеричном виде их легко запомнить.

Эти константы очень часто используются для определения отдельных бит как флагов. Например, это из файла `ssl_private.h` из исходников Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE           (0)
#define SSL_OPT_RELSET         (1<<0)
#define SSL_OPT_STDENVVARS     (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH  (1<<4)
#define SSL_OPT_STRICTREQUIRE  (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Вернемся назад к нашему примеру.

Макрос `IS_SET` проверяет наличие этого бита в a. Макрос `IS_SET` на самом деле это операция логического И (*AND*) и она возвращает 0 если бита там нет, либо эту же битовую маску, если бит там есть. В Си/Си++, конструкция `if()` срабатывает, если выражение внутри её не ноль, пусть хоть 123456, поэтому все будет работать.

17.4.1. x86

MSVC

Компилируем (MSVC 2010):

Листинг 17.6: MSVC 2010

```
_rt$ = -8           ; size = 4
_i$ = -4            ; size = 4
_a$ = 8             ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]   ; инкремент i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; цикл закончился?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f              ; результат исполнения инструкции AND был 0?
                                     ; тогда пропускаем следующие команды
    mov    eax, DWORD PTR _rt$[ebp]  ; нет, не ноль
    add    eax, 1                    ; инкремент rt
    mov    DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp    SHORT $LN3@f
$LN2@f:
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

17.4.2. x64

Немного изменим пример, расширив его до 64-х бит:

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

Оптимизирующий MSVC 2010

Листинг 17.7: MSVC 2010

```
a$ = 8
f       PROC
; RCX = входное значение
        xor     eax, eax
        mov     edx, 1
```

```
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f:
        test    rdx, rcx
; не было такого бита во входном значении?
; тогда пропустить следующую инструкцию INC.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Здесь используется инструкция ROL вместо SHL, которая на самом деле «rotate left» (прокручивать влево) вместо «shift left» (сдвиг влево), но здесь, в этом примере, она работает так же как и SHL.

R8 здесь считает от 64 до 0. Это как бы инвертированная переменная i.

Вот таблица некоторых регистров в процессе исполнения:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

Оптимизирующий MSVC 2012

Листинг 17.8: MSVC 2012

```
a$ = 8
f       PROC
; RCX = входное значение
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f:
; проход 1 ---------------------------------
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -------------------------------------------
; проход 2 ---------------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax     ; rt++
$LN11@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -------------------------------------------
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Оптимизирующий MSVC 2012 делает почти то же самое что и оптимизирующий MSVC 2010, но почему-то он генерирует 2 идентичных тела цикла и счетчик цикла теперь 32 вместо 64. Честно говоря, нельзя сказать, почему. Какой-то трюк с оптимизацией? Может быть, телу цикла лучше быть немного длиннее? Так или иначе, такой код здесь уместен, чтобы показать, что результат компилятора иногда может быть очень странный и нелогичный, но прекрасно работающий, конечно же.

17.5. Вывод

Инструкции сдвига, аналогичные операторам Си/Си++ $\ll$ и $\gg$, в x86 это SHR/SHL (для беззнаковых значений), SAR/SHL (для знаковых значений).

Инструкции сдвига в ARM это LSR/LSL (для беззнаковых значений), ASR/LSL (для знаковых значений). Можно также добавлять суффикс сдвига для некоторых инструкций (которые называются «data processing instructions»).

17.5.1. Проверка определенного бита (известного на стадии компиляции)

Проверить, присутствует ли бит 1000000 (0x40) в значении в регистре:

Листинг 17.9: Си/Си++

```
if (input&0x40)
    ...
```

Листинг 17.10: x86

```
TEST REG, 40h
JNZ is_set
; бит не установлен
```

Листинг 17.11: x86

```
TEST REG, 40h
JZ is_cleared
; бит установлен
```

Иногда AND используется вместо TEST, но флаги выставляются точно также.

17.5.2. Проверка определенного бита (заданного во время исполнения)

Это обычно происходит при помощи вот такого фрагмента на Си/Си++ (сдвинуть значение на n бит вправо, затем отрезать самый младший бит):

Листинг 17.12: Си/Си++

```
if ((value>>n)&1)
    ....
```

Это обычно реализуется в x86-коде так:

Листинг 17.13: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Или (сдвинуть 1 n раз влево, изолировать этот же бит во входном значении и проверить, не ноль ли он):

Листинг 17.14: Си/Си++

```
if (value & (1<<n))
    ....
```

Это обычно так реализуется в x86-коде:

Листинг 17.15: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

ГЛАВА 17. РАБОТА С ОТДЕЛЬНЫМИ БИТАМИ

17.5.3. Установка определенного бита (известного во время компиляции)

Листинг 17.16: Си/Си++

```
value=value|0x40;
```

Листинг 17.17: x86

```
OR REG, 40h
```

17.5.4. Установка определенного бита (заданного во время исполнения)

Листинг 17.18: Си/Си++

```
value=value|(1<<n);
```

Это обычно так реализуется в x86-коде:

Листинг 17.19: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

17.5.5. Сброс определенного бита (известного во время компиляции)

Просто исполните операцию логического «И» (AND) с инвертированным значением:

Листинг 17.20: Си/Си++

```
value=value&(~0x40);
```

Листинг 17.21: x86

```
AND REG, 0FFFFFFBFh
```

Листинг 17.22: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Это на самом деле сохранение всех бит кроме одного.

17.5.6. Сброс определенного бита (заданного во время исполнения)

Листинг 17.23: Си/Си++

```
value=value&(~(1<<n));
```

Листинг 17.24: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

Глава 18

Линейный конгруэнтный генератор как генератор псевдослучайных чисел

Линейный конгруэнтный генератор, пожалуй, самый простой способ генерировать псевдослучайные числа. Он не в почете в наше время[1], но он настолько прост (только одно умножение, одно сложение и одна операция «И»), что мы можем использовать его в качестве примера.

```
#include <stdint.h>

// константы из книги Numerical Recipes
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

Здесь две функции: одна используется для инициализации внутреннего состояния, а вторая вызывается собственно для генерации псевдослучайных чисел.

Мы видим что в алгоритме применяются две константы. Они взяты из [Pre+07]. Определим их используя выражение Си/Си++ `#define`. Это макрос. Разница между макросом в Си/Си++ и константой в том, что все макросы заменяются на значения препроцессором Си/Си++ и они не занимают места в памяти как переменные. А константы, напротив, это переменные только для чтения. Можно взять указатель (или адрес) переменной-константы, но это невозможно сделать с макросом.

Последняя операция «И» нужна, потому что согласно стандарту Си `my_rand()` должна возвращать значение в пределах 0..32767. Если вы хотите получать 32-битные псевдослучайные значения, просто уберите последнюю операцию «И».

18.1. x86

Листинг 18.1: Оптимизирующий MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
```

[1] Вихрь Мерсенна куда лучше

```
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767              ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

Вот мы это и видим: обе константы встроены в код. Память для них не выделяется. Функция `my_srand()` просто копирует входное значение во внутреннюю переменную `rand_state`.

`my_rand()` берет её, вычисляет следующее состояние `rand_state`, обрезает его и оставляет в регистре EAX.

Неоптимизированная версия побольше:

Листинг 18.2: Неоптимизирующий MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767              ; 00007fffH
        pop     ebp
        ret     0
_rand   ENDP

_TEXT   ENDS
```

18.2. x64

Версия для x64 почти такая же, и использует 32-битные регистры вместо 64-битных (потому что мы работаем здесь с переменными типа *int*). Но функция `my_srand()` берет входной аргумент из регистра ECX, а не из стека:

Листинг 18.3: Оптимизирующий MSVC 2013 x64

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
```

```
; ECX = входной аргумент
        mov     DWORD PTR rand_state, ecx
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525      ; 0019660dH
        add     eax, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767                              ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Глава 19

Структуры

В принципе, структура в Си/Си++ это, с некоторыми допущениями, просто всегда лежащий рядом, и в той же последовательности, набор переменных, не обязательно одного типа [1].

19.1. MSVC: Пример SYSTEMTIME

Возьмем, к примеру, структуру SYSTEMTIME[2] из win32 описывающую время.

Она объявлена так:

Листинг 19.1: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Пишем на Си функцию для получения текущего системного времени:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Что в итоге (MSVC 2010):

Листинг 19.2: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
```

[1] AKA «гетерогенный контейнер»
[2] MSDN: SYSTEMTIME structure

```
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push   eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Под структуру в стеке выделено 16 байт — именно столько будет `sizeof(WORD)*8` (в структуре 8 переменных с типом WORD).

Обратите внимание на тот факт, что структура начинается с поля `wYear`. Можно сказать, что в качестве аргумента для `GetSystemTime()`[3]передается указатель на структуру SYSTEMTIME, но можно также сказать, что передается указатель на поле `wYear`, что одно и тоже! `GetSystemTime()` пишет текущий год в тот WORD на который указывает переданный указатель, затем сдвигается на 2 байта вправо, пишет текущий месяц, и т.д., и т.д.

19.1.1. Замена структуры массивом

Тот факт, что поля структуры — это просто переменные расположенные рядом, легко проиллюстрировать следующим образом. Глядя на описание структуры SYSTEMTIME, можно переписать этот простой пример так:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
};
```

Компилятор немного ворчит:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

Тем не менее, выдает такой код:

Листинг 19.3: Неоптимизирующий MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16   ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
```

[3]MSDN: GetSystemTime function

```
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

И это работает так же!

Любопытно что результат на ассемблере неотличим от предыдущего . Таким образом, глядя на этот код, никогда нельзя сказать с уверенностью, была ли там объявлена структура, либо просто набор переменных.

Тем не менее, никто в здравом уме делать так не будет. Потому что это неудобно. К тому же, иногда, поля в структуре могут меняться разработчиками, переставляться местами, и т.д.

19.2. Выделяем место для структуры через malloc()

Однако, бывает и так, что проще хранить структуры не в стеке, а в куче:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Скомпилируем на этот раз с оптимизацией (/Ox) чтобы было проще увидеть то, что нам нужно.

Листинг 19.4: Оптимизирующий MSVC

```
_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
```

```
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP
```

Итак, `sizeof(SYSTEMTIME) = 16`, именно столько байт выделяется при помощи `malloc()`. Она возвращает указатель на только что выделенный блок памяти в EAX, который копируется в `ESI`. Win32 функция `GetSystemTime()` обязуется сохранить состояние `ESI`, поэтому здесь оно нигде не сохраняется и продолжает использоваться после вызова `GetSystemTime()`.

Новая инструкция — `MOVZX` (*Move with Zero eXtend*). Она нужна почти там же где и `MOVSX`, только всегда очищает остальные биты в 0. Дело в том, что `printf()` требует 32-битный тип *int*, а в структуре лежит WORD — это 16-битный беззнаковый тип. Поэтому копируя значение из WORD в *int*, нужно очистить биты от 16 до 31, иначе там будет просто случайный мусор, оставшийся от предыдущих действий с регистрами.

В этом примере можно также представить структуру как массив 8-и WORD-ов:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Получим такое:

Листинг 19.5: Оптимизирующий MSVC

```
$SG78594 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
```

```
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

И снова мы получаем идентичный код, неотличимый от предыдущего. Но и снова нужно отметить, что в реальности так лучше не делать, если только вы не знаете точно, что вы делаете.

19.3. Упаковка полей в структуре

Достаточно немаловажный момент, это упаковка полей в структурах[4].

Возьмем простой пример:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

Как видно, мы имеем два поля *char* (занимающий один байт) и еще два — *int* (по 4 байта).

19.3.1. x86

Компилируется это все в:

Листинг 19.6: MSVC 2012 /GS- /Ob0

```
_tmp$ = -16
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    mov    BYTE PTR _tmp$[ebp], 1     ; установить поле a
```

[4] См. также: Wikipedia: Выравнивание данных

```
7      mov    DWORD PTR _tmp$[ebp+4], 2  ; установить поле b
8      mov    BYTE PTR _tmp$[ebp+8], 3   ; установить поле c
9      mov    DWORD PTR _tmp$[ebp+12], 4 ; установить поле d
10     sub    esp, 16                    ; выделить место для временной структуры
11     mov    eax, esp
12     mov    ecx, DWORD PTR _tmp$[ebp]  ; скопировать нашу структуру во временную
13     mov    DWORD PTR [eax], ecx
14     mov    edx, DWORD PTR _tmp$[ebp+4]
15     mov    DWORD PTR [eax+4], edx
16     mov    ecx, DWORD PTR _tmp$[ebp+8]
17     mov    DWORD PTR [eax+8], ecx
18     mov    edx, DWORD PTR _tmp$[ebp+12]
19     mov    DWORD PTR [eax+12], edx
20     call    _f
21     add    esp, 16
22     xor    eax, eax
23     mov    esp, ebp
24     pop    ebp
25     ret    0
26 _main    ENDP
27
28 _s$ = 8 ; size = 16
29 ?f@@YAXUs@@@Z PROC ; f
30     push   ebp
31     mov    ebp, esp
32     mov    eax, DWORD PTR _s$[ebp+12]
33     push   eax
34     movsx  ecx, BYTE PTR _s$[ebp+8]
35     push   ecx
36     mov    edx, DWORD PTR _s$[ebp+4]
37     push   edx
38     movsx  eax, BYTE PTR _s$[ebp]
39     push   eax
40     push   OFFSET $SG3842
41     call   _printf
42     add    esp, 20
43     pop    ebp
44     ret    0
45 ?f@@YAXUs@@@Z ENDP ; f
46 _TEXT    ENDS
```

Кстати, мы передаем всю структуру, но в реальности, как видно, структура в начале копируется во временную структуру (выделение места под нее в стеке происходит в строке 10, а все 4 поля, по одному, копируются в строках 12 … 19), затем передается только указатель на нее (или адрес). Структура копируется, потому что неизвестно, будет ли функция `f()` модифицировать структуру или нет. И если да, то структура внутри `main()` должна остаться той же. Мы могли бы использовать указатели на Си/Си++, и итоговый код был бы почти такой же, только копирования не было бы.

Мы видим здесь что адрес каждого поля в структуре выравнивается по 4-байтной границе. Так что каждый *char* здесь занимает те же 4 байта что и *int*. Зачем? Затем что процессору удобнее обращаться по таким адресам и кэшировать данные из памяти.

Но это не экономично по размеру данных.

Попробуем скомпилировать тот же исходник с опцией (`/Zp1`) (*/Zp[n] pack structures on n-byte boundary*).

Листинг 19.7: MSVC 2012 /GS- /Zp1

```
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    mov    BYTE PTR _tmp$[ebp], 1     ; установить поле a
    mov    DWORD PTR _tmp$[ebp+1], 2  ; установить поле b
    mov    BYTE PTR _tmp$[ebp+5], 3   ; установить поле c
    mov    DWORD PTR _tmp$[ebp+6], 4  ; установить поле d
    sub    esp, 12                    ; выделить место для временной структуры
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; скопировать 10 байт
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
```

ГЛАВА 19. СТРУКТУРЫ

```
    mov    cx, WORD PTR _tmp$[ebp+8]
    mov    WORD PTR [eax+8], cx
    call    _f
    add    esp, 12
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_TEXT    SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC    ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+6]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+5]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+1]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP    ; f
```

Теперь структура занимает 10 байт и все *char* занимают по байту. Что это дает? Экономию места. Недостаток — процессор будет обращаться к этим полям не так эффективно по скорости, как мог бы.

Структура так же копируется в `main()`. Но не по одному полю, а 10 байт, при помощи трех пар MOV. Почему не 4? Компилятор рассудил, что будет лучше скопировать 10 байт при помощи 3 пар MOV, чем копировать два 32-битных слова и два байта при помощи 4 пар MOV.

Как нетрудно догадаться, если структура используется много в каких исходниках и объектных файлах, все они должны быть откомпилированы с одним и тем же соглашением об упаковке структур.

Помимо ключа MSVC `/Zp`, указывающего, по какой границе упаковывать поля структур, есть также опция компилятора `#pragma pack`, её можно указывать прямо в исходнике. Это справедливо и для MSVC[5] и GCC[6].

Давайте теперь вернемся к `SYSTEMTIME`, которая состоит из 16-битных полей. Откуда наш компилятор знает что их надо паковать по однобайтной границе?

В файле `WinNT.h` попадается такое:

Листинг 19.8: WinNT.h

```
#include "pshpack1.h"
```

И такое:

Листинг 19.9: WinNT.h

```
#include "pshpack4.h"                   // 4 byte packing is the default
```

Сам файл PshPack1.h выглядит так:

Листинг 19.10: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
```

[5] MSDN: Working with Packing Structures
[6] Structure-Packing Pragmas

```
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Собственно, так и задается компилятору, как паковать объявленные после `#pragma pack` структуры.

19.3.2. Еще кое-что

Передача структуры как аргумент функции (вместо передачи указателя на структуру) это то же что и передача всех полей структуры по одному. Если поля в структуре пакуются по умолчанию, то функцию f() можно переписать так:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

И в итоге будет такой же код.

19.4. Вложенные структуры

Теперь, как насчет ситуаций, когда одна структура определена внутри другой структуры?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

…в этом случае, оба поля `inner_struct` просто будут располагаться между полями a,b и d,e в `outer_struct`.

Компилируем (MSVC 2010):

Листинг 19.11: Оптимизирующий MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT    SEGMENT
```

```
_s$ = 8
_f    PROC
    mov    eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f    ENDP

_s$ = -24
_main    PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
    pop    ebx
    add    esp, 24
    ret    0
_main    ENDP
```

Очень любопытный момент в том, что глядя на этот код на ассемблере, мы даже не видим, что была использована какая-то еще другая структура внутри этой! Так что, пожалуй, можно сказать, что все вложенные структуры в итоге разворачиваются в одну, *линейную* или *одномерную* структуру.

Конечно, если заменить объявление `struct inner_struct c;` на `struct inner_struct *c;` (объявляя таким образом указатель), ситуация будет совсем иная.

19.5. Работа с битовыми полями в структуре

19.5.1. Пример CPUID

Язык Си/Си++ позволяет указывать, сколько именно бит отвести для каждого поля структуры. Это удобно если нужно экономить место в памяти. К примеру, для переменной типа *bool* достаточно одного бита. Но, это не очень удобно, если нужна скорость.

Рассмотрим пример с инструкцией CPUID[7]. Эта инструкция возвращает информацию о том, какой процессор имеется в наличии и какие возможности он имеет.

Если перед исполнением инструкции в EAX будет 1, то CPUID вернет упакованную в EAX такую информацию о процессоре:

3:0 (4 бита)	Stepping
7:4 (4 бита)	Model
11:8 (4 бита)	Family
13:12 (2 бита)	Processor Type
19:16 (4 бита)	Extended Model
27:20 (8 бита)	Extended Family

MSVC 2010 имеет макрос для CPUID, а GCC 4.4.1 — нет. Поэтому для GCC сделаем эту функцию сами, используя его встроенный ассемблер[8].

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

После того как CPUID заполнит EAX/EBX/ECX/EDX, у нас они отразятся в массиве b[]. Затем, мы имеем указатель на структуру CPUID_1_EAX, и мы указываем его на значение EAX из массива b[].

Иными словами, мы трактуем 32-битный *int* как структуру. Затем мы читаем отдельные биты из структуры.

[7] wikipedia
[8] Подробнее о встроенном ассемблере GCC

MSVC

Компилируем в MSVC 2008 с опцией /Ox:

Листинг 19.12: Оптимизирующий MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
    and    edx, 15
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 16
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call   _printf

    shr    esi, 20
    and    esi, 255
    push   esi
    push   OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call   _printf
    add    esp, 48
    pop    esi

    xor    eax, eax
    pop    ebx

    add    esp, 16
    ret    0
_main    ENDP
```

ГЛАВА 19. СТРУКТУРЫ

Инструкция SHR сдвигает значение из EAX на то количество бит, которое нужно *пропустить*, то есть, мы игнорируем некоторые биты *справа*.

А инструкция AND очищает биты *слева* которые нам не нужны, или же, говоря иначе, она оставляет по маске только те биты в EAX, которые нам сейчас нужны.

Глава 20

64-битные значения в 32-битной среде

20.1. Возврат 64-битного значения

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

20.1.1. x86

64-битные значения в 32-битной среде возвращаются из функций в паре регистров EDX:EAX.

Листинг 20.1: Оптимизирующий MSVC 2010

```
_f      PROC
        mov     eax, -1867788817        ; 90abcdefH
        mov     edx, 305419896          ; 12345678H
        ret     0
_f      ENDP
```

20.2. Передача аргументов, сложение, вычитание

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

 ГЛАВА 20. 64-БИТНЫЕ ЗНАЧЕНИЯ В 32-БИТНОЙ СРЕДЕ

20.2.1. x86

Листинг 20.2: Оптимизирующий MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f_add  ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff_subH
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f_sub  ENDP
```

В `f_add_test()` видно, как каждое 64-битное число передается двумя 32-битными значениями, сначала старшая часть, затем младшая.

Сложение и вычитание происходит также парами.

При сложении, в начале складываются младшие 32 бита. Если при сложении был перенос, выставляется флаг CF. Следующая инструкция ADC складывает старшие части чисел, но также прибавляет единицу если $CF = 1$.

Вычитание также происходит парами. Первый SUB может также включить флаг переноса CF, который затем будет проверяться в SBB : если флаг переноса включен, то от результата отнимется единица .

Легко увидеть, как результат работы `f_add()` затем передается в `printf()`.

20.3. Умножение, деление

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
```

```
};
```

20.3.1. x86

Листинг 20.3: Оптимизирующий MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; long long multiplication (умножение значений типа long long)
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; unsigned long long division (деление беззнаковых значений типа long long)
        pop     ebp
        ret     0
_f_div  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; unsigned long long remainder (вычисление беззнакового остатка)
        pop     ebp
        ret     0
_f_rem  ENDP
```

Умножение и деление — это более сложная операция, так что обычно, компилятор встраивает вызовы библиотечных функций, делающих это.

20.4. Сдвиг вправо

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

20.4.1. x86

Листинг 20.4: Оптимизирующий MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

Сдвиг происходит также в две операции: в начале сдвигается младшая часть, затем старшая . Но младшая часть сдвигается при помощи инструкции SHRD, она сдвигает значение в EDX на 7 бит, но подтягивает новые биты из EAX, т.е. из старшей части. Старшая часть сдвигается более известной инструкцией SHR: действительно, ведь освободившиеся биты в старшей части нужно просто заполнить нулями.

20.5. Конвертирование 32-битного значения в 64-битное

```
#include <stdint.h>

int64_t f (int32_t a)
{
        return a;
};
```

20.5.1. x86

Листинг 20.5: Оптимизирующий MSVC 2012

```
_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP
```

Здесь появляется необходимость расширить 32-битное знаковое значение в 64-битное знаковое. Конвертировать беззнаковые значения очень просто: нужно просто выставить в 0 все биты в старшей части . Но для знаковых типов это не подходит: знак числа должен быть скопирован в старшую часть числа-результата . Здесь это делает инструкция CDQ, она берет входное значение в EAX, расширяет его до 64-битного, и оставляет его в паре регистров EDX:EAX . Иными словами, инструкция CDQ узнает знак числа в EAX (просто берет самый старший бит в EAX) и в зависимости от этого, выставляет все 32 бита в EDX в 0 или в 1. Её работа в каком-то смысле напоминает работу инструкции MOVSX.

Глава 21

64 бита

21.1. x86-64

Это расширение x86-архитуктуры до 64 бит.

С точки зрения начинающего reverse engineer-а, наиболее важные отличия от 32-битного x86 это:

- Почти все регистры (кроме FPU и SIMD) расширены до 64-бит и получили префикс R-. И еще 8 регистров добавлено. В итоге имеются эти GPR-ы: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

  К ним также можно обращаться так же, как и прежде. Например, для доступа к младшим 32 битам RAX можно использовать EAX:

7 (номер байта)	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

  У новых регистров R8-R15 также имеются их *младшие части*: R8D-R15D (младшие 32-битные части), R8W-R15W (младшие 16-битные части), R8L-R15L (младшие 8-битные части).

7 (номер байта)	6	5	4	3	2	1	0
R8							
				R8D			
						R8W	
							R8L

  Удвоено количество SIMD-регистров: с 8 до 16: XMM0-XMM15.

- В win64 передача всех параметров немного иная, это немного похоже на fastcall . Первые 4 аргумента записываются в регистры RCX, RDX, R8, R9, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт чтобы вызываемая функция могла сохранить там первые 4 аргумента и использовать эти регистры по своему усмотрению. Короткие функции могут использовать аргументы прямо из регистров, но бóльшие функции могут сохранять их значения на будущее.

  Соглашение System V AMD64 ABI (Linux, *BSD, Mac OS X)[Mit13] также напоминает fastcall, использует 6 регистров RDI, RSI, RDX, RCX, R8, R9 для первых шести аргументов. Остальные передаются через стек.

- *int* в Си/Си++ остается 32-битным для совместимости.

- Все указатели теперь 64-битные.

  На это иногда сетуют: ведь теперь для хранения всех указателей нужно в 2 раза больше места в памяти, в т.ч. и в кэш-памяти, не смотря на то что x64-процессоры могут адресовать только 48 бит внешней RAM[1].

Из-за того, что регистров общего пользования теперь вдвое больше, у компиляторов теперь больше свободного места для маневра, называемого register allocation. Для нас это означает, что в итоговом коде будет меньше локальных переменных.

Кстати, существуют процессоры с еще большим количеством GPR, например, Itanium — 128 регистров.

[1] Random-access memory

Часть II

Важные фундаментальные вещи



ГЛАВА 22. ПРЕДСТАВЛЕНИЕ ЗНАКА В ЧИСЛАХ

Глава 22

Представление знака в числах

Методов представления чисел с знаком «плюс» или «минус» несколько[1], но в компьютерах обычно применяется метод «дополнительный код» или «two's complement».

Вот таблица некоторые значений байтов:

двоичное	шестнадцатеричное	беззнаковое	знаковое (дополнительный код)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000110	0x6	6	6
00000101	0x5	5	5
00000100	0x4	4	4
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
11111100	0xfc	252	-4
11111011	0xfb	251	-5
11111010	0xfa	250	-6
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

Разница в подходе к знаковым/беззнаковым числам, собственно, нужна потому что, например, если представить 0xFFFFFFFE и 0x0000002 как беззнаковое, то первое число (4294967294) больше второго (2). Если их оба представить как знаковые, то первое будет -2, которое, разумеется, меньше чем второе (2). Вот почему инструкции для условных переходов (11 (стр. 34)) представлены в обоих версиях — и для знаковых сравнений (например, JG, JL) и для беззнаковых (JA, JB).

Для простоты, вот что нужно знать:

- Числа бывают знаковые и беззнаковые.
- Знаковые типы в Си/Си++:
  - `int64_t` (-9,223,372,036,854,775,808..9,223,372,036,854,775,807) (- 9.2.. 9.2 квинтиллионов) или `0x8000000000000000..0x7FFFFFFFFFFFFFFF`),
  - *int* (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) или `0x80000000..0x7FFFFFFF`),
  - *char* (-128..127 или `0x80..0x7F`),
  - `ssize_t`.

  Беззнаковые:
  - `uint64_t` (0..18,446,744,073,709,551,615 (18 квинтиллионов) или `0..0xFFFFFFFFFFFFFFFF`),
  - `unsigned int` (0..4,294,967,295 (4.3Gb) или `0..0xFFFFFFFF`),

[1] wikipedia

  - `unsigned char` (0..255 или `0..0xFF`),
  - `size_t`.
- У знаковых чисел знак определяется самым старшим битом: 1 означает «минус», 0 означает «плюс» .
- Преобразование в бóльшие типы данных обходится легко: 20.5 (стр. 101).
- Изменить знак легко: просто инвертируйте все биты и прибавьте 1. Мы можем заметить, что число другого знака находится на другой стороне на том же расстоянии от нуля. Прибавление единицы необходимо из-за присутствия нуля посредине.
- Инструкции сложения и вычитания работают одинаково хорошо и для знаковых и для беззнаковых значений . Но для операций умножения и деления, в x86 имеются разные инструкции : `IDIV`/`IMUL` для знаковых и `DIV`/`MUL` для беззнаковых.

Глава 23

Память

Есть три основных типа памяти:

- Глобальная память. AKA «static memory allocation». Нет нужды явно выделять, выделение происходит просто при объявлении переменных/массивов глобально. Это глобальные переменные расположенные в сегменте данных или констант. Доступны глобально (поэтому считаются анти-паттерном). Не удобны для буферов/массивов, потому что должны иметь фиксированный размер. Переполнения буфера, случающиеся здесь, обычно перезаписывают переменные или буферы расположенные рядом в памяти. Пример в этой книге: 7.2 (стр. 20).
- Стек. AKA «allocate on stack», «выделить память в/на стеке». Выделение происходит просто при объявлении переменных/массивов локально в функции. Обычно это локальные для функции переменные. Иногда эти локальные переменные также доступны и для нисходящих функций (callee-функциям, если функция-caller передает указатель на переменную в функцию-callee). Выделение и освобождение очень быстрое, достаточно просто сдвига SP. Но также не удобно для буферов/массивов, потому что размер буфера фиксирован, если только не используется `alloca()` (5.2.4 (стр. 11)) (или массив с переменной длиной). Переполнение буфера обычно перезаписывает важные структуры стека: 16.2 (стр. 63).
- Куча (*heap*). AKA «dynamic memory allocation», «выделить память в куче». Выделение происходит при помощи вызова `malloc()/free()` или `new/delete` в Си++. Самый удобный метод: размер блока может быть задан во время исполнения. Изменение размера возможно (при помощи `realloc()`), но может быть медленным. Это самый медленный метод выделения памяти: аллокатор памяти должен поддерживать и обновлять все управляющие структуры во время выделения и освобождения. Переполнение буфера обычно перезаписывает все эти структуры. Выделения в куче также ведут к проблеме утечек памяти: каждый выделенный блок должен быть явно освобожден, но кто-то может забыть об этом, или делать это неправильно. Еще одна проблема — это «использовать после освобождения» — использовать блок памяти после того как `free()` был вызван на нем, это тоже очень опасно. Пример в этой книге: 19.2 (стр. 88).

Часть III

Поиск в коде того что нужно

Современное ПО, в общем-то, минимализмом не отличается.

Но не потому, что программисты слишком много пишут, а потому что к исполняемым файлам обыкновенно прикомпилируют все подряд библиотеки. Если бы все вспомогательные библиотеки всегда выносили во внешние DLL, мир был бы иным. (Еще одна причина для Си++ — STL и прочие библиотеки шаблонов.)

Таким образом, очень полезно сразу понимать, какая функция из стандартной библиотеки или более-менее известной (как Boost[1], libpng[2]), а какая — имеет отношение к тому что мы пытаемся найти в коде.

Переписывать весь код на Си/Си++, чтобы разобраться в нем, безусловно, не имеет никакого смысла.

Одна из важных задач reverse engineer-а это быстрый поиск в коде того что собственно его интересует.

Дизассемблер IDA позволяет делать поиск как минимум строк, последовательностей байт, констант. Можно даже сделать экспорт кода в текстовый файл .lst или .asm и затем натравить на него `grep`, awk, и т.д.

Когда вы пытаетесь понять, что делает тот или иной код, это запросто может быть какая-то опенсорсная библиотека вроде libpng. Поэтому, когда находите константы, или текстовые строки, которые выглядят явно знакомыми, всегда полезно их *погуглить*. А если вы найдете искомый опенсорсный проект где это используется, то тогда будет достаточно будет просто сравнить вашу функцию с ней. Это решит часть проблем.

К примеру, если программа использует какие-то XML-файлы, первым шагом может быть установление, какая именно XML-библиотека для этого используется, ведь часто используется какая-то стандартная (или очень известная) вместо самодельной.

К примеру, автор этих строк однажды пытался разобраться как происходит компрессия/декомпрессия сетевых пакетов в SAP 6.0. Это очень большая программа, но к ней идет подробный .PDB-файл с отладочной информацией, и это очень удобно. Он в конце концов пришел к тому что одна из функций декомпрессирующая пакеты называется CsDecomprLZC(). Не сильно раздумывая, он решил погуглить и оказалось, что функция с таким же названием имеется в MaxDB (это опенсорсный проект SAP) .

`http://www.google.com/search?q=CsDecomprLZC`

Каково же было мое удивление, когда оказалось, что в MaxDB используется точно такой же алгоритм, скорее всего, с таким же исходником.

[1] http://go.yurichev.com/17036
[2] http://go.yurichev.com/17037

Глава 24

Связь с внешним миром (win32)

Иногда, чтобы понять что делает та или иная функция, можно её не разбирать, а просто посмотреть на её входы и выходы. Так можно сэкономить время.

Обращения к файлам и реестру: для самого простого анализа может помочь утилита Process Monitor[1] от SysInternals.

Для анализа обращения программы к сети, может помочь Wireshark[2].

Затем всё-таки придётся смотреть внутрь.

Первое на что нужно обратить внимание, это какие функции из API[3] ОС и какие функции стандартных библиотек используются.

Если программа поделена на главный исполняемый файл и группу DLL-файлов, то имена функций в этих DLL, бывает так, могут помочь.

Если нас интересует, что именно приводит к вызову `MessageBox()` с определенным текстом, то первое что можно попробовать сделать: найти в сегменте данных этот текст, найти ссылки на него, и найти, откуда может передаться управление к интересующему нас вызову `MessageBox()`.

Если речь идет о компьютерной игре, и нам интересно какие события в ней более-менее случайны, мы можем найти функцию `rand()` или её заменитель (как алгоритм Mersenne twister), и посмотреть, из каких мест эта функция вызывается и что самое главное: как используется результат этой функции.

Но если это не игра, а `rand()` используется, то также весьма любопытно, зачем. Бывают неожиданные случаи вроде использования `rand()` в алгоритме для сжатия данных (для имитации шифрования): blog.yurichev.com.

24.1. Часто используемые функции Windows API

Это функции которые можно увидеть в числе импортируемых. Но также нельзя забывать, что далеко не все они были использованы в коде написанном автором. Немалая часть может вызываться из библиотечных функций и CRT-кода.

- Работа с реестром (advapi32.dll): RegEnumKeyEx[4] [5], RegEnumValue[6] [5], RegGetValue[7] [5], RegOpenKeyEx[8] [5], RegQueryValueEx[9] [5].
- Работа с текстовыми .ini-файлами (kernel32.dll): GetPrivateProfileString [10] [5].
- Диалоговые окна (user32.dll): MessageBox [11] [5], MessageBoxEx [12] [5], SetDlgItemText [13] [5], GetDlgItemText [14] [5].
- Работа с ресурсами : (user32.dll): LoadMenu [15] [5].

[1] http://go.yurichev.com/17301
[2] http://go.yurichev.com/17303
[3] Application programming interface
[4] MSDN
[5] Может иметь суффикс -A для ASCII-версии и -W для Unicode-версии
[6] MSDN
[7] MSDN
[8] MSDN
[9] MSDN
[10] MSDN
[11] MSDN
[12] MSDN
[13] MSDN
[14] MSDN
[15] MSDN

- Работа с TCP/IP-сетью (ws2_32.dll): WSARecv [16], WSASend [17].
- Работа с файлами (kernel32.dll): CreateFile [18] [5], ReadFile [19], ReadFileEx [20], WriteFile [21], WriteFileEx [22].
- Высокоуровневая работа с Internet (wininet.dll): WinHttpOpen [23].
- Проверка цифровой подписи исполняемого файла (wintrust.dll): WinVerifyTrust [24].
- Стандартная библиотека MSVC (в случае динамического связывания) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

24.2. tracer: Перехват всех функций в отдельном модуле

В tracer есть поддержка точек останова INT3, хотя и срабатывающие только один раз, но зато их можно установить на все сразу функции в некоей DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Либо, поставим INT3-прерывание на все функции, имена которых начинаются с префикса `xml`:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

В качестве обратной стороны медали, такие прерывания срабатывают только один раз.

Tracer покажет вызов какой-либо функции, если он случится, но только один раз. Еще один недостаток — увидеть аргументы функции также нельзя.

Тем не менее, эта возможность очень удобна для тех ситуаций, когда вы знаете что некая программа использует некую DLL, но не знаете какие именно функции в этой DLL. И функций много.

Например, попробуем узнать, что использует cygwin-утилита uptime:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Так мы можем увидеть все функции из библиотеки cygwin1.dll, которые были вызваны хотя бы один раз, и откуда:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3))
```

[16] MSDN
[17] MSDN
[18] MSDN
[19] MSDN
[20] MSDN
[21] MSDN
[22] MSDN
[23] MSDN
[24] MSDN

Глава 25

Строки

25.1. Текстовые строки

25.1.1. Си/Си++

Обычные строки в Си заканчиваются нулем (ASCIIZ-строки).

Причина, почему формат строки в Си именно такой (оканчивающийся нулем) вероятно историческая . В [Rit79] мы можем прочитать:

> A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

Строки выглядят в Hiew или FAR Manager точно так же :

```
int main()
{
        printf ("Hello, world!\n");
};
```

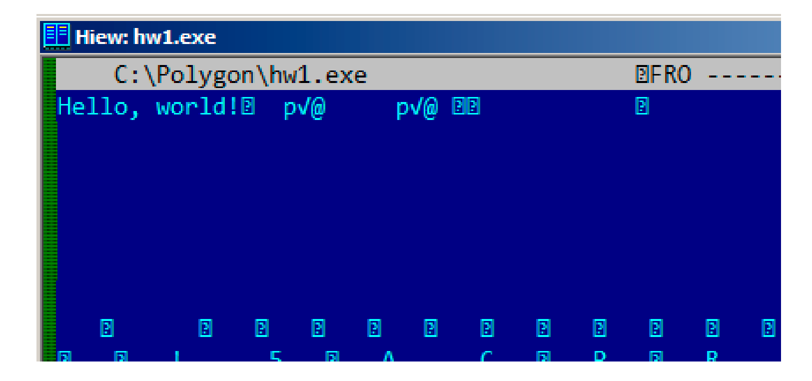


Рис. 25.1: Hiew

25.1.2. Borland Delphi

Когда кодируются строки в Pascal и Delphi, сама строка предваряется 8-битным или 32-битным значением, в котором закодирована длина строки.

Например:

Листинг 25.1: Delphi

```
CODE:00518AC8                 dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0

...
```

```
CODE:00518AFC                 dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

25.1.3. Unicode

Нередко уникодом называют все способы передачи символов, когда символ занимает 2 байта или 16 бит . Это распространенная терминологическая ошибка. Уникод — это стандарт, присваивающий номер каждому символу многих письменностей мира, но не описывающий способ кодирования.

Наиболее популярные способы кодирования: UTF-8 (наиболее часто используется в Интернете и *NIX-системах) и UTF-16LE (используется в Windows).

UTF-8

UTF-8 это один из очень удачных способов кодирования символов. Все символы латиницы кодируются так же, как и в ASCII-кодировке, а символы, выходящие за пределы ASCII-7-таблицы, кодируются несколькими байтами. 0 кодируется, как и прежде, нулевыми байтом, так что все стандартные функции Си продолжают работать с UTF-8-строками так же как и с обычными строками.

Посмотрим, как символы из разных языков кодируются в UTF-8 и как это выглядит в FAR, в кодировке 437 [1]:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

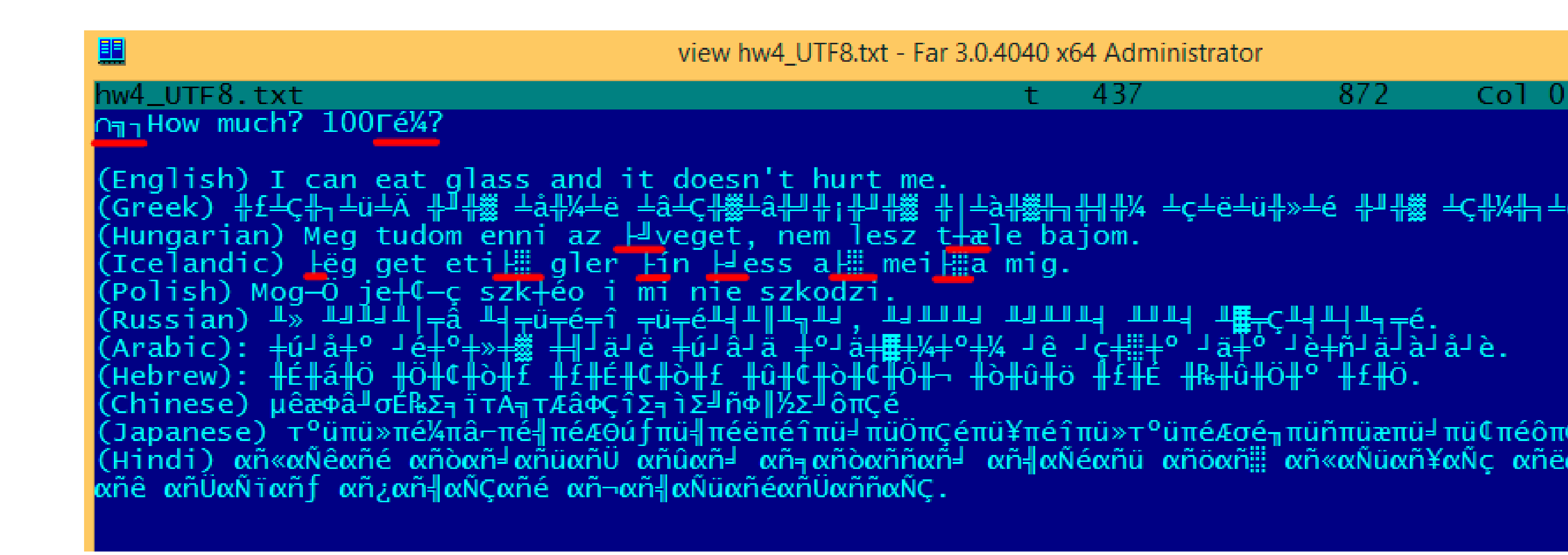


Рис. 25.2: FAR: UTF-8

Видно, что строка на английском языке выглядит точно так же, как и в ASCII-кодировке . В венгерском языке используются латиница плюс латинские буквы с диакритическими знаками . Здесь видно, что эти буквы кодируются несколькими байтами, они подчеркнуты красным . То же самое с исландским и польским языками. В самом начале имеется также символ валюты «Евро», который кодируется тремя байтами. Остальные системы письма здесь никак не связаны с латиницей . По крайней мере о русском, арабском, иврите и хинди мы можем сказать, что здесь видны повторяющиеся байты, что

[1] Пример и переводы на разные языки были взяты здесь: http://go.yurichev.com/17304

не удивительно, ведь, обычно буквы из одной и той же системы письменности расположены в одной или нескольких таблицах уникода, поэтому часто их коды начинаются с одних и тех же цифр .

В самом начале, перед строкой «How much?», видны три байта, которые на самом деле BOM[2] . BOM показывает, какой способ кодирования будет сейчас использоваться.

UTF-16LE

Многие функции win32 в Windows имееют суффикс -A и -W. Первые функции работают с обычными строками, вторые с UTF-16LE-строками (*wide*). Во втором случае, каждый символ обычно хранится в 16-битной переменной типа *short* .

Строки с латинскими буквами выглядят в Hiew или FAR как перемежающиеся с нулевыми байтами :

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

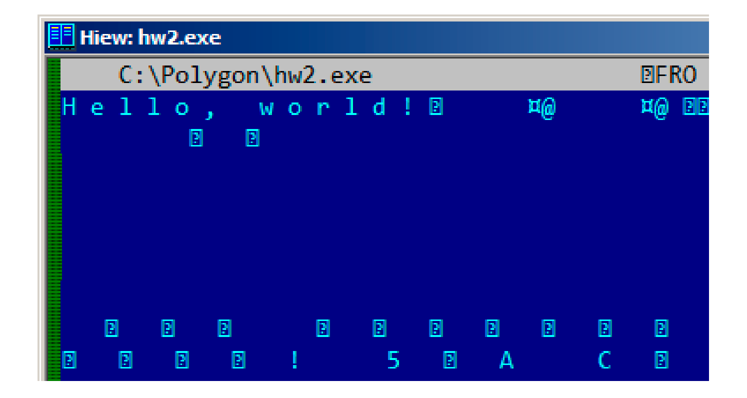


Рис. 25.3: Hiew

Подобное можно часто увидеть в системных файлах Windows NT:

Рис. 25.4: Hiew

В IDA, уникодом называется именно строки с символами, занимающими 2 байта:

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Вот как может выглядеть строка на русском языке («И снова здравствуйте!») закодированная в UTF-16LE:

[2] Byte order mark

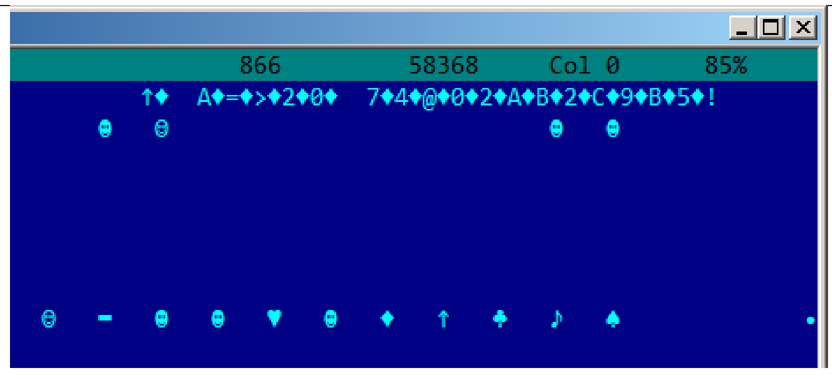


Рис. 25.5: Hiew: UTF-16LE

То что бросается в глаза — это то что символы перемежаются ромбиками (который имеет код 4) . Действительно, в уникоде кириллиные символы находятся в четвертом блоке [3]. Таким образом, все кириллиные символы в UTF-16LE находятся в диапазоне 0x400-0x4FF.

Вернемся к примеру, где одна и та же строка написана разными языками. Здесь посмотрим в кодировке UTF-16LE.

Рис. 25.6: FAR: UTF-16LE

Здесь мы также видим BOM в самом начале. Все латинские буквы перемежаются с нулевыми байтом. Некоторые буквы с диакритическими знаками (венгерский и исландский языки) также подчеркнуты красным.

[3] wikipedia

25.1.4. Base64

Кодировка base64 очень популярна в тех случаях, когда нужно передать двоичные данные как текстовую строку. По сути, этот алгоритм кодирует 3 двоичных байта в 4 печатаемых символа: все 26 букв латинского алфавита (в обоих регистрах), цифры, знак плюса («+») и слэша («/»), в итоге это получается 64 символа.

Одна отличительная особенность строк в формате base64, это то что они часто (хотя и не всегда) заканчиваются одним или двумя символами знака равенства («=») для выравнивания, например:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=
```

```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuQ==
```

Так что знак равенства («=») никогда не встречается в середине строк закодированных в base64.

25.2. Сообщения об ошибках и отладочные сообщения

Очень сильно помогают отладочные сообщения, если они имеются. В некотором смысле, отладочные сообщения, это отчет о том, что сейчас происходит в программе. Зачастую, это `printf()`-подобные функции, которые пишут куда-нибудь в лог, а бывает так что и не пишут ничего, но вызовы остались, так как эта сборка — не отладочная, а *release*. Если в отладочных сообщениях дампятся значения некоторых локальных или глобальных переменных, это тоже может помочь, как минимум, узнать их имена. Например, в Oracle RDBMS одна из таких функций: `ksdwrt()`.

Осмысленные текстовые строки вообще очень сильно могут помочь. Дизассемблер IDA может сразу указать, из какой функции и из какого её места используется эта строка. Встречаются и смешные случаи [4].

Сообщения об ошибках также могут помочь найти то что нужно. В Oracle RDBMS сигнализация об ошибках проходит при помощи вызова некоторой группы функций.
Тут еще немного об этом : blog.yurichev.com.

Можно довольно быстро найти, какие функции сообщают о каких ошибках, и при каких условиях. Это, кстати, одна из причин, почему в защите софта от копирования, бывает так, что сообщение об ошибке заменяется невнятным кодом или номером ошибки. Мало кому приятно, если взломщик быстро поймет, из-за чего именно срабатывает защита от копирования, просто по сообщению об ошибке.

25.3. Подозрительные магические строки

Некоторые магические строки, используемые в бэкдорах выглядят очень подозрительно. Например, в домашних роутерах TP-Link WR740 был бэкдор [5]. Бэкдор активировался при посещении следующего URL:
`http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html`.
Действительно, строка «userRpmNatDebugRpm26525557» присутствует в прошивке. Эту строку нельзя было нагуглить до распространения информации о бэкдоре. Вы не найдете ничего такого ни в одном RFC[6]. Вы не найдете ни одного алгоритма, который бы использовал такие странные последовательности байт. И это не выглядит как сообщение об ошибке, или отладочное сообщение. Так что проверить использование подобных странных строк — это всегда хорошая идея.

Иногда такие строки кодируются при помощи base64[7]. Так что неплохая идея их всех декодировать и затем просмотреть глазами, пусть даже бегло.

Более точно, такой метод сокрытия бэкдоров называется «security through obscurity» (безопасность через запутанность).

[4] blog.yurichev.com
[5] `http://sekurak.pl/tp-link-httptftp-backdoor/`, на русском: `http://m.habrahabr.ru/post/172799/`
[6] Request for Comments
[7] Например, бэкдор в кабельном модеме Arris: `http://www.securitylab.ru/analytics/461497.php`

Глава 26

Вызовы assert()

Может также помочь наличие `assert()` в коде: обычно этот макрос оставляет название файла-исходника, номер строки, и условие.

Наиболее полезная информация содержится в assert-условии, по нему можно судить по именам переменных или именам полей структур. Другая полезная информация — это имена файлов, по их именам можно попытаться предположить, что там за код. Также, по именам файлов можно опознать какую-либо очень известную опен-сорсную библиотеку.

Листинг 26.1: Пример информативных вызовов assert()

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Полезно «гуглить» и условия и имена файлов, это может вывести вас к опен-сорсной бибилотеке. Например, если «погуглить» «sp->lzw_nbits <= BITS_MAX», это вполне предсказуемо выводит на опенсорсный код, что-то связанное с LZW-компрессией.

Глава 27

Константы

Люди, включая программистов, часто используют круглые числа вроде 10, 100, 1000, в т.ч. и в коде.

Практикующие реверсеры, обычно, хорошо знают их в шестнадцатеричном представлении : 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Иногда попадаются константы 0xAAAAAAAA (10101010101010101010101010101010) и 0x55555555 (01010101010101010101010101010101) — это чередующиеся биты. Это помогает отличить некоторый сигнал от сигнала где все биты включены (1111 …) или выключены (0000 …). Например, константа 0x55AA используется как минимум в бут-секторе, MBR[1], и в ПЗУ[2] плат-расширений IBM-компьютеров.

Некоторые алгоритмы, особенно криптографические, используют хорошо различимые константы, которые при помощи IDA легко находить в коде.

Например, алгоритм MD5[3] инициализирует свои внутренние переменные так:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Если в коде найти использование этих четырех констант подряд — очень высокая вероятность что эта функция имеет отношение к MD5.

Еще такой пример это алгоритмы CRC16/CRC32, часто, алгоритмы вычисления контрольной суммы по CRC используют заранее заполненные таблицы, вроде:

Листинг 27.1: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        ...
```

27.1. Magic numbers

Немало форматов файлов определяет стандартный заголовок файла где используются *magic number*[4], один или даже несколько.

Скажем, все исполняемые файлы для Win32 и MS-DOS начинаются с двух символов «MZ»[5].

В начале MIDI-файла должно быть «MThd». Если у нас есть использующая для чего-нибудь MIDI-файлы программа очень вероятно, что она будет проверять MIDI-файлы на правильность хотя бы проверяя первые 4 байта.

Это можно сделать при помощи:

(*buf* указывает на начало загруженного в память файла)

[1] Master Boot Record
[2] Постоянное запоминающее устройство
[3] wikipedia
[4] wikipedia
[5] wikipedia

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...либо вызвав функцию сравнения блоков памяти `memcmp()` или любой аналогичный код, вплоть до инструкции CMPSB .

Найдя такое место мы получаем как минимум информацию о том, где начинается загрузка MIDI-файла, во-вторых, мы можем увидеть где располагается буфер с содержимым файла, и что еще оттуда берется, и как используется.

27.1.1. DHCP

Это касается также и сетевых протоколов. Например, сетевые пакеты протокола DHCP содержат так называемую *magic cookie*: `0x63538263`. Какой-либо код, генерирующий пакеты по протоколу DHCP где-то и как-то должен внедрять в пакет также и эту константу. Найдя её в коде мы сможем найти место где происходит это и не только это. Любая программа, получающая DHCP-пакеты, должна где-то как-то проверять *magic cookie*, сравнивая это поле пакета с константой.

Например, берем файл dhcpcore.dll из Windows 7 x64 и ищем эту константу. И находим, два раза: оказывается, эта константа используется в функциях с красноречивыми названиями `DhcpExtractOptionsForValidation()` и `DhcpExtractFullO`

Листинг 27.2: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ; DATA XREF: ↙
    ↳ DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ; DATA XREF: DhcpExtractFullOptions+97
```

А вот те места в функциях где происходит обращение к константам:

Листинг 27.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

И:

Листинг 27.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

27.2. Поиск констант

В IDA это очень просто, Alt-B или Alt-I. А для поиска константы в большом количестве файлов, либо для поиска их в неисполняемых файлах, имеется небольшая утилита *binary grep*[6].

[6]GitHub

Глава 28

Поиск нужных инструкций

Если программа использует инструкции сопроцессора, и их не очень много, то можно попробовать вручную проверить отладчиком какую-то из них.

К примеру, нас может заинтересовать, при помощи чего Microsoft Excel считает результаты формул, введенных пользователем. Например, операция деления.

Если загрузить excel.exe (из Office 2010) версии 14.0.4756.1000 в IDA, затем сделать полный листинг и найти все инструкции FDIV (но кроме тех, которые в качестве второго операнда используют константы — они, очевидно, не подходят нам):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

…то окажется, что их всего 144.

Мы можем вводить в Excel строку вроде =(1/3) и проверить все эти инструкции.

Проверяя каждую инструкцию в отладчике или tracer (проверять эти инструкции можно по 4 за раз), окажется, что нам везет и срабатывает всего лишь 14-я по счету:

```
.text:3011E919 DC 33                                fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

В ST(0) содержится первый аргумент (1), второй содержится в [EBX].

Следующая за FDIV инструкция (FSTP) записывает результат в память:

```
.text:3011E91B DD 1E                                fstp    qword ptr [esi]
```

Если поставить breakpoint на ней, то мы можем видеть результат:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

А также, в рамках пранка[1], модифицировать его на лету:

[1] practical joke

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel показывает в этой ячейке 666, что окончательно убеждает нас в том, что мы нашли нужное место.

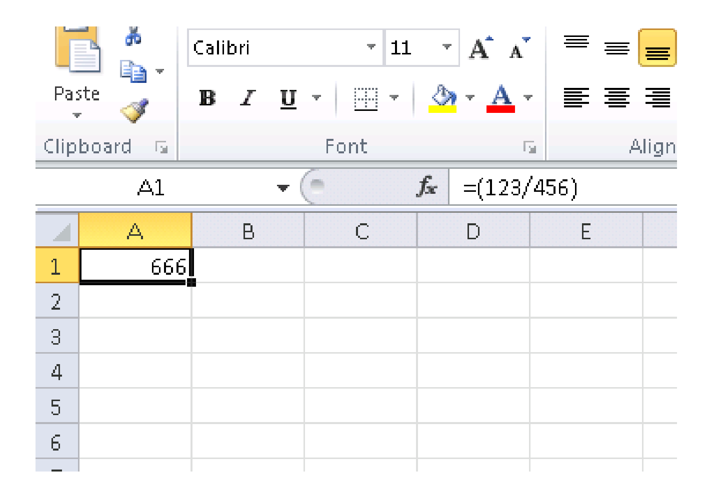


Рис. 28.1: Пранк сработал

Если попробовать ту же версию Excel, только x64, то окажется что там инструкций FDIV всего 12, причем нужная нам — третья по счету.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Видимо, все дело в том, что много операций деления переменных типов *float* и *double* компилятор заменил на SSE-инструкции вроде DIVSD, коих здесь теперь действительно много (DIVSD присутствует в количестве 268 инструкций).

Глава 29

Подозрительные паттерны кода

29.1. Инструкции XOR

Инструкции вроде XOR op, op (например, XOR EAX, EAX) обычно используются для обнуления регистра, однако, если операнды разные, то применяется операция именно «исключающего или». Эта операция очень редко применяется в обычном программировании, но применяется очень часто в криптографии, включая любительскую. Особенно подозрительно, если второй операнд — это большое число. Это может указывать на шифрование, вычисление контрольной суммы, и т.д.

Этот AWK-скрипт можно использовать для обработки листингов (.lst) созданных IDA :

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp") { ↙
    ↘ print $1, $2, tmp, ",", $4 } }' filename.lst
```

29.2. Вручную написанный код на ассемблере

Современные компиляторы не генерируют инструкции LOOP и RCL. С другой стороны, эти инструкции хорошо знакомы кодерам, предпочитающим писать прямо на ассемблере. Если такие инструкции встретились, можно сказать с какой-то вероятностью, что этот фрагмент кода написан вручную.

Также, пролог/эпилог функции обычно не встречается в ассемблерном коде, написанном вручную.

Как правило, в вручную написанном коде, нет никакого четкого метода передачи аргументов в функцию .

Пример из ядра Windows 2003 (файл ntoskrnl.exe):

```
MultiplyTest proc near              ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                         ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                      ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near              ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
             mov     eax, 417A000h
             mul     ecx
             cmp     edx, 2
             stc
             jnz     short locret_62057F
             cmp     eax, 0FE7A000h
             stc
             jnz     short locret_62057F
```

```
             clc
locretі_62057F:                        ; CODE XREF: Multiply+10
                                       ; Multiply+18
             retn
Multiply     endp
```

Действительно, если заглянуть в исходные коды WRK[1] v1.2, данный код можно найти в файле *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

[1]Windows Research Kernel

 ГЛАВА 30. ИСПОЛЬЗОВАНИЕ MAGIC NUMBERS ДЛЯ ТРАССИРОВКИ

Глава 30

Использование magic numbers для трассировки

Нередко бывает нужно узнать, как используется то или иное значение, прочитанное из файла либо взятое из пакета, принятого по сети. Часто, ручное слежение за нужной переменной это трудный процесс. Один из простых методов (хотя и не полностью надежный на 100%) это использование вашей собственной *magic number*.

Это чем-то напоминает компьютерную томографию: пациенту перед сканированием вводят в кровь рентгеноконтрастный препарат, хорошо отсвечивающий в рентгеновских лучах. Известно, как кровь нормального человека расходится, например, по почкам, и если в этой крови будет препарат, то при томографии будет хорошо видно, достаточно ли хорошо кровь расходится по почкам и нет ли там камней, например, и прочих образований.

Мы можем взять 32-битное число вроде `0x0badf00d`, либо чью-то дату рождения вроде `0x11101979` и записать это, занимающее 4 байта число, в какое-либо место файла используемого исследуемой нами программой.

Затем, при трассировки этой программы, в том числе, при помощи tracer в режиме *code coverage*, а затем при помощи *grep* или простого поиска по текстовому файлу с результатами трассировки, мы можем легко увидеть, в каких местах кода использовалось это значение, и как.

Пример результата работы tracer в режиме *cc*, к которому легко применить утилиту *grep*:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Это справедливо также и для сетевых пакетов. Важно только, чтобы наш *magic number* был как можно более уникален и не присутствовал в самом коде.

Помимо tracer, такой эмулятор MS-DOS как DosBox, в режиме heavydebug, может писать в отчет информацию обо всех состояниях регистра на каждом шаге исполнения программы[1], так что этот метод может пригодиться и для исследования программ под DOS.

[1]См. также мой пост в блоге об этой возможности в DosBox: blog.yurichev.com

Глава 31

Прочее

31.1. Общая идея

Нужно стараться как можно чаще ставить себя на место программиста и задавать себе вопрос, как бы вы сделали ту или иную вещь в этом случае и в этой программе.

31.2. Некоторые паттерны в бинарных файлах

Иногда мы можем легко заметить массив 16/32/64-битных значений визуально, в шестнадцатеричном редакторе. Вот пример очень типичного MIPS-кода. Как мы наверное помним, каждая инструкция в MIPS (а также в ARM в режиме ARM, или ARM64) имеет длину 32 бита (или 4 байта), так что такой код это массив 32-битных значений. Глядя на этот скриншот, можно увидеть некий узор. Вертикальные красные линии добавлены для ясности:

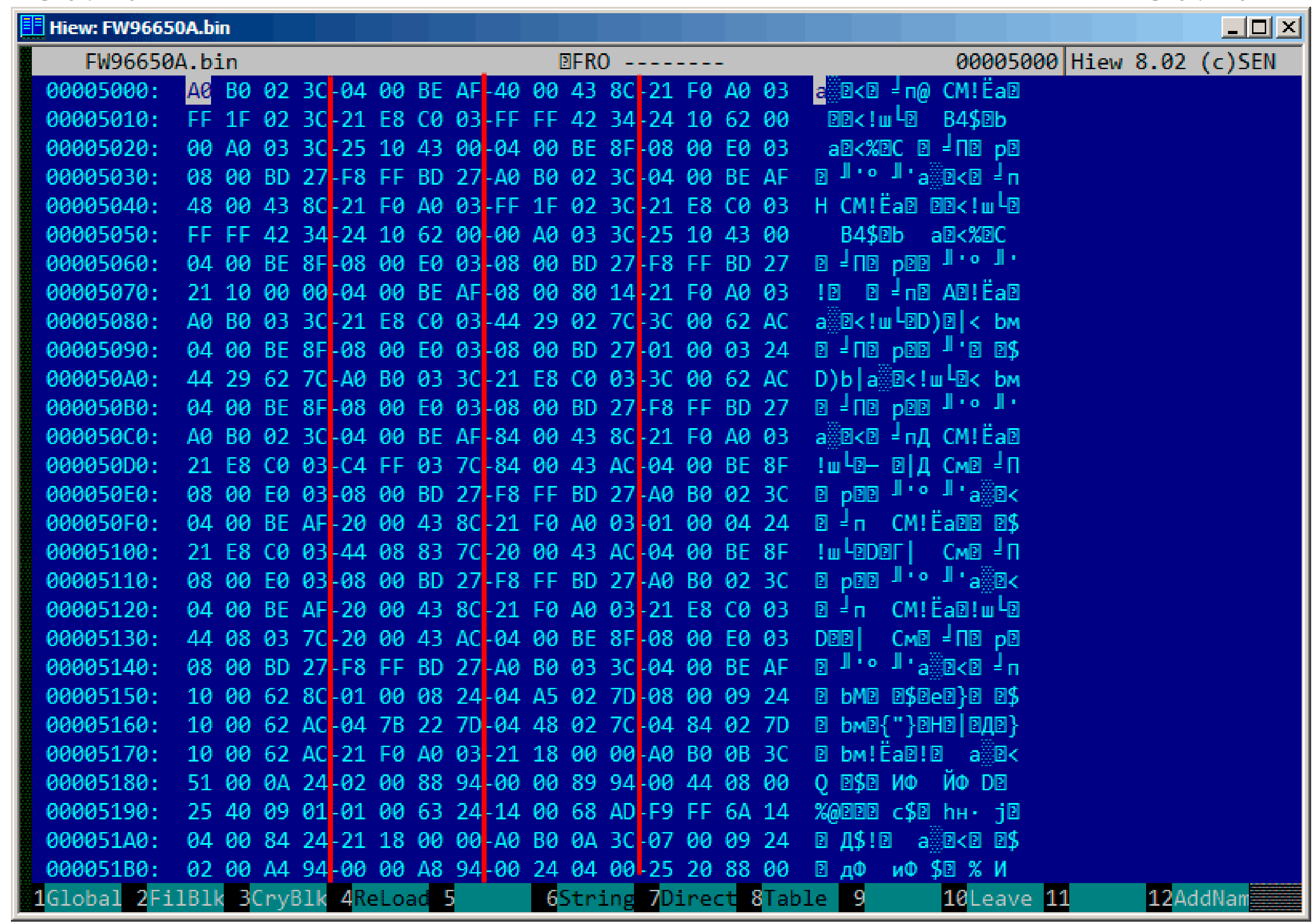


Рис. 31.1: Hiew: очень типичный код для MIPS

31.3. Сравнение «снимков» памяти

Метод простого сравнения двух снимков памяти для поиска изменений часто применялся для взлома игр на 8-битных компьютерах и взлома файлов с записанными рекордными очками.

К примеру, если вы имеете загруженную игру на 8-битном компьютере (где самой памяти не очень много, но игра занимает еще меньше), и вы знаете что сейчас у вас, условно, 100 пуль, вы можете сделать «снимок» всей памяти и сохранить где-то. Затем просто стреляете куда угодно, у вас станет 99 пуль, сделать второй «снимок», и затем сравнить эти два снимка: где-то наверняка должен быть байт, который в начале был 100, а затем стал 99. Если учесть, что игры на тех маломощных домашних компьютерах обычно были написаны на ассемблере и подобные переменные там были глобальные, то можно с уверенностью сказать, какой адрес в памяти всегда отвечает за количество пуль. Если поискать в дизассемблированном коде игры все обращения по этому адресу, несложно найти код, отвечающий за уменьшение пуль и записать туда инструкцию NOP или несколько NOP-в, так мы получим игру в которой у игрока всегда будет 100 пуль, например. А так как игры на тех домашних 8-битных компьютерах всегда загружались по одним и тем же адресам, и версий одной игры редко когда было больше одной продолжительное время, то геймеры-энтузиасты знали, по какому адресу (используя инструкцию языка BASIC POKE) что записать после загрузки игры, чтобы хакнуть её. Это привело к появлению списков «читов» состоящих из инструкций POKE, публикуемых в журналах посвященным 8-битным играм. См. также: wikipedia.

Точно так же легко модифицировать файлы с сохраненными рекордами (кто сколько очков набрал), впрочем, это может сработать не только с 8-битными играми. Нужно заметить, какой у вас сейчас рекорд и где-то сохранить файл с очками. Затем, когда очков станет другое количество, просто сравнить два файла, можно даже DOS-утилитой FC[1] (файлы рекордов, часто, бинарные). Где-то будут отличаться несколько байт, и легко будет увидеть, какие именно отвечают за количество очков. Впрочем, разработчики игр полностью осведомлены о таких хитростях и могут защититься от этого.

[1]утилита MS-DOS для сравнения двух файлов побайтово

 ГЛАВА 31. ПРОЧЕЕ

31.3.1. Реестр Windows

А еще можно вспомнить сравнение реестра Windows до инсталляции программы и после . Это также популярный метод поиска, какие элементы реестра программа использует. Наверное это причина, почему так популярны shareware-программы для очистки реестра в Windows.

31.3.2. Блинк-компаратор

Сравнение файлов или слепков памяти вообще, немного напоминает блинк-компаратор [2]: устройство, которое раньше использовали астрономы для поиска движущихся небесных объектов. Блинк-компаратор позволял быстро переключаться между двух отснятых в разное время кадров, и астроном мог увидеть разницу визуально. Кстати, при помощи блинк-компаратора, в 1930 был открыт Плутон.

[2] http://go.yurichev.com/17349

Часть IV

Инструменты

Глава 32

Дизассемблер

32.1. IDA

Старая бесплатная версия доступна для скачивания [1].

[1] hex-rays.com/products/ida/support/download_freeware.shtml

Глава 33

Отладчик

33.1. tracer

Автор часто использует *tracer*[1] вместо отладчика.

Со временем, автор этих строк отказался использовать отладчик, потому что всё что ему нужно от него это иногда подсмотреть какие-либо аргументы какой-либо функции во время исполнения или состояние регистров в определенном месте. Каждый раз загружать отладчик для этого это слишком, поэтому родилась очень простая утилита *tracer*. Она консольная, запускается из командной строки, позволяет перехватывать исполнение функций, ставить точки останова на произвольные места, смотреть состояние регистров, модифицировать их, и т.д.

Но для учебы очень полезно трассировать код руками в отладчике, наблюдать как меняются значения регистров (например, как минимум классический SoftICE, OllyDbg, WinDbg подсвечивают измененные регистры), флагов, данные, менять их самому, смотреть реакцию, и т.д.

[1] yurichev.com

Глава 34

Декомпиляторы

Пока существует только один публично доступный декомпилятор в Си высокого качества : Hex-Rays:
hex-rays.com/products/decompiler/

Глава 35

Прочие инструменты

- Microsoft Visual Studio Express[1]: Усеченная бесплатная версия Visual Studio, пригодная для простых экспериментов.
- Hiew[2] для мелкой модификации кода в исполняемых файлах.
- binary grep: небольшая утилита для поиска констант (либо просто последовательности байт) в большом количестве файлов, включая неисполняемые: GitHub.

[1] visualstudio.com/en-US/products/visual-studio-express-vs
[2] hiew.ru

Часть V

Что стоит почитать

Глава 36

Книги

36.1. Windows

[RA09].

36.2. Си/Си++

[ISO13].

36.3. x86 / x86-64

[Int13], [AMD13]

36.4. ARM

Документация от ARM: http://go.yurichev.com/17024

36.5. Криптография

[Sch94]

Глава 37

Блоги

37.1. Windows

- Microsoft: Raymond Chen
- nynaeve.net

Глава 38

Прочее

Имеются два отличных субреддита на reddit.com посвященных RE[1] : reddit.com/r/ReverseEngineering/ и reddit.com/r/remath (для тем посвященных пересечению RE и математики).

Имеется также часть сайта Stack Exchange посвященная RE: reverseengineering.stackexchange.com.

На IRC есть канал ##re на FreeNode[2].

[1] Reverse Engineering
[2] freenode.net

Послесловие

Глава 39

Вопросы?

Совершенно по любым вопросам вы можете не раздумывая писать автору: <dennis(a)yurichev.com>

Есть идеи о том, что ещё можно добавить в эту книгу?

Пожалуйста, присылайте мне информацию о замеченных ошибках (включая грамматические), и т.д.

Автор много работает над книгой, поэтому номера страниц, листингов, и т.д. очень часто меняются. Пожалуйста, в своих письмах мне не ссылайтесь на номера страниц и листингов. Есть метод проще: сделайте скриншот страницы, затем в графическом редакторе подчеркните место, где вы видите ошибку, и отправьте автору. Так он может исправить её намного быстрее. Ну а если вы знакомы с git и LaTeX, вы можете исправить ошибку прямо в исходных текстах: GitHub.

Не бойтесь побеспокоить меня написав мне о какой-то мелкой ошибке, даже если вы не очень уверены. Я всё-таки пишу для начинающих, поэтому мнение и коментарии именно начинающих очень важны для моей работы.

Внимание: это сокращенная LITE-версия!

Она примерно в 6 раз короче полной версии (~150 страниц) и предназначена для тех, кто хочет краткого введения в основы reverse engineering. Здесь нет ничего о MIPS, ARM, OllyDBG, GCC, GDB, IDA, нет задач, примеров, и т.д.

Если вам всё ещё интересен reverse engineering, полная версия книги всегда доступна на моем сайте: beginners.re.

Список принятых сокращений

ОС Операционная Система . . . ix

ЯП Язык Программирования . . . 3

ПЗУ Постоянное запоминающее устройство . . . 118

RA Адрес возврата . . . 45

SP stack pointer. SP/ESP/RSP в x86/x64. SP в ARM. . . . 9

PC Program Counter. IP/EIP/RIP в x86/64. PC в ARM. . . . 142

IDA Интерактивный дизассемблер и отладчик, разработан Hex-Rays

MSVC Microsoft Visual C++

AKA Also Known As (Также известный как)

CRT C runtime library . . . 6

CPU Central processing unit . . . ix

SIMD Single instruction, multiple data . . . 53

ISA Instruction Set Architecture (Архитектура набора команд) . . . 3

SEH Structured Exception Handling . . . 12

NOP No OPeration . . . 25

RAM Random-access memory . . . 102

API Application programming interface . . . 110

ASCIIZ ASCII Zero (ASCII-строка заканчивающаяся нулем) . . . 24

VM Virtual Memory (виртуальная память)

WRK Windows Research Kernel . . . 123

GPR General Purpose Registers (регистры общего пользования) . . . 3

RE Reverse Engineering . . . 136

BOM Byte order mark . . . 114

MBR Master Boot Record . . . 118

RFC Request for Comments . . . 116

EOF End of file (конец файла) . . . 22

Glossary

вещественное число числа, которые могут иметь точку. в Си/Си++ это *float* и *double* . 61

декремент Уменьшение на 1. 52, 57

инкремент Увеличение на 1. 52, 57

произведение Результат умножения. 28

указатель стека Регистр указывающий на место в стеке . 6, 9, 11, 14, 141

частное Результат деления. 61

anti-pattern Нечто широко известное как плохое решение . 20, 107

callee Вызываемая функция. 17, 28, 107

caller Функция вызывающая другую функцию. 29, 107

heap (куча) обычно, большой кусок памяти предоставляемый ОС, так что прикладное ПО может делить его как захочет. malloc()/free() работают с кучей . 9, 11, 88

jump offset Часть опкода JMP или Jcc инструкции, просто прибавляется к адресу следующей инструкции, и так вычисляется новый PC[1]. Может быть отрицательным. 37

NOP «no operation», холостая инструкция. 126

PDB (Win32) Файл с отладочной информацией, обычно просто имена функций, но иногда имена аргументов функций и локальных переменных . 109

POKE Инструкция языка BASIC записывающая байт по определенному адресу . 126

register allocator Функция компилятора распределяющая локальные переменные по регистрам процессора . 57, 102

reverse engineering процесс понимания как устроена некая вещь, иногда, с целью клонирования оной . iv

stack frame Часть стека, в которой хранится информация, связанная с текущей функцией: локальные переменные, аргументы функции, RA, и т.д.. 18, 28

stdout standard output. 12, 45

tracer Моя простейшая утилита для отладки. Читайте больше об этом тут : 33.1 (стр. 130). 111, 120, 124

Windows NT Windows NT, 2000, XP, Vista, 7, 8. 114

[1] Program Counter. IP/EIP/RIP в x86/64. PC в ARM.

Предметный указатель

Литература

[AMD13] AMD. AMD64 Architecture Programmer's Manual. Также доступно здесь: http://go.yurichev.com/17284. 2013.

[Dij68] Edsger W. Dijkstra. «Letters to the editor: go to statement considered harmful». В: Commun. ACM 11.3 (март 1968), с. 147—148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: http://go.yurichev.com/17299.

[Fog13] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs / An optimization guide for assembly programmers and con http://go.yurichev.com/17278. 2013.

[Int13] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C. Также доступно здесь: http://go.yurichev.com/17283. 2013.

[ISO07] ISO. ISO/IEC 9899:TC3 (C C99 standard). Также доступно здесь: http://go.yurichev.com/17274. 2007.

[ISO13] ISO. ISO/IEC 14882:2011 (C++ 11 standard). Также доступно здесь: http://go.yurichev.com/17275. 2013.

[Ker88] Brian W. Kernighan. The C Programming Language. Под ред. Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[Knu74] Donald E. Knuth. «Structured Programming with go to Statements». В: ACM Comput. Surv. 6.4 (дек. 1974). Also available as http://go.yurichev.com/17271, с. 261—301. ISSN: 0360-0300. DOI: 10.1145/356635.356640. URL: http://go.yurichev.com/17300.

[Knu98] Donald E. Knuth. The Art of Computer Programming Volumes 1-3 Boxed Set. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201485419.

[Mit13] Michael Matz / Jan Hubicka / Andreas Jaeger / Mark Mitchell. System V Application Binary Interface. AMD64 Architecture Pr Также доступно здесь: http://go.yurichev.com/17295. 2013.

[One96] Aleph One. «Smashing The Stack For Fun And Profit». В: Phrack (1996). Также доступно здесь: http://go.yurichev.com/17266.

[Pre+07] William H. Press и др. Numerical Recipes. 2007.

[RA09] Mark E. Russinovich и David A. Solomon with Alex Ionescu. Windows® Internals: Including Windows Server 2008 and Wind 2009.

[Rit79] Dennis M. Ritchie. «The Evolution of the Unix Time-sharing System». В: (1979).

[RT74] D. M. Ritchie и K. Thompson. «The UNIX Time Sharing System». В: (1974). Также доступно здесь: http://go.yurichev.com/17270.

[Sch94] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 1994.

[Str13] Bjarne Stroustrup. The C++ Programming Language, 4th Edition. 2013.

[Yur13] Dennis Yurichev. C/C++ programming language notes. Также доступно здесь: http://go.yurichev.com/17289. 2013.